DEVELOPMENT AND VALIDATION OF SOFTWARE QUALITY PREDICTION MODELS USING MACHINE LEARNING TECHNIQUES

A Thesis Submitted
In Partial Fulfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

By

MADHUKAR CHERUKURI
(2K18/PHDCO/16)

Under the Supervision of
Prof. Ruchika Malhotra
Head of Department
Department of Software Engineering

Department of Software Engineering

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

November, 2024

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Main Bawana Road, Delhi-42

CANDIDATE'S DECLARATION

I **Madhukar Cherukuri** hereby certify that the work which is being presented in the thesis entitled **"Development and Validation of Software Quality Prediction Models using Machine Learning Techniques"** in partial fulfillment of the requirements for the award of the Degree of Doctor of Philosophy, submitted in the Department of **Software Engineering**, Delhi Technological University is an authenticate record of my own work carried out during the period from **2018** to **2024** under the supervision of **Prof. Ruchika Malhotra.**

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.

Candidate's Signature

This is to certify that the student has incorporated all the corrections suggested by the examiners 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

Certified that **Mr. Madhukar Cherukuri (Roll No. 2K18/PHDCO/16)** has carried out his research work presented in this thesis entitled **"Development and Validation of Software Quality Prediction Models using Machine Learning Techniques"** for the award of **Doctor of Philosophy** from Department of Software Engineering, Delhi Technological University, Delhi, under my supervision. The thesis embodies results of original work, and studies are carried out by the student himself and the contents of the thesis do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

Signature

PROF. RUCHIKA MALHOTRA

HoD and DRC Chairperson

Department of Software Engineering

Delhi Technological University, Delhi 110042

Date:

”This thesis is dedicated to my beloved grandmother, Late Smt. Kanaka Durga, and to my dearest friend, Sri Vishnu, both of whom have profoundly shaped my life through their wisdom and moral guidance. Their unwavering support and desire to see me succeed have inspired me to pursue my dreams with passion and integrity.”

Acknowledgment

The completion of my Ph.D. has been an incredible journey of learning, discovery, and perseverance. This journey would not have been possible without the guidance, support, and encouragement of many individuals to whom I am deeply grateful.

First and foremost, I would like to express my deepest gratitude to my supervisor, **Prof. Ruchika Malhotra**. Her unwavering support and insightful guidance were invaluable throughout my Ph.D. journey. From the initial stages of shaping the research problem to the final stages of completion, her intellectual rigor and expertise helped me stay focused and motivated. She provided not only academic guidance but also continuous encouragement, making me believe in the significance of my work even when I faced challenges. Her dedication to her students is truly inspiring, and I am fortunate to have had her as my mentor.

I would like to express my heartfelt appreciation to the Hon'ble Vice Chancellor, **Prof. Prateek Sharma** whose leadership and vision have fostered a supportive academic environment that allowed my research to thrive. I am also deeply thankful to **Prof. Yogesh Singh**, Hon'ble Vice Chancellor, University of Delhi whose initiatives laid the foundation for academic excellence and provided valuable resources during his tenure as Vice Chancellor of Delhi Technological University. Additionally, I express my heartfelt appreciation to **Dr. Kamal Pathak**, Registrar, Guru Gobind Singh Indraprastha University and **Prof. Rajeshwari Pandey**, Controller of Examination for their constant support, understanding, and flexibility offered throughout my Ph.D. journey. I would also like to thank the **faculty members and staff** of the Department of Software Engineering and Department of Computer Science & Engineering for their assistance and support during my time there. The departments have been a stimulating and enriching environment that has nurtured my research.

I am greatly indebted to my parents, **Mr. Venkateswara Rao** and **Mrs. Sita Ratna Kumari** whose sacrifices and values have shaped who I am today. I express my heartfelt appreciation to my wife **Mrs. Daneesha and my extended family- my parents-in-law, uncles, aunts, brothers and sisters** for their unending support and for always cheering me on. Their unwavering support, encouragement, and motivation

have been the key to all of my accomplishments in life. I am also grateful to my **friends and colleagues** for the countless discussions, the much-needed breaks, and the continuous moral support.

To my beloved daughter **Kushvi**, you are the brightest light in my life. Your laughter, curiosity, and boundless energy have been a constant source of joy and inspiration. Every moment spent with you has reminded me of the pure beauty and wonder in the world, even amidst the challenges of this journey.

Finally, I would like to acknowledge everyone who, in one way or another, contributed to the successful completion of this work. Your encouragement and support, whether big or small, have helped me reach this significant milestone in my life. Thank you all.

Madhukar Cherukuri

ABSTRACT

Software quality plays a pivotal role in ensuring the success of software projects, especially in today's rapidly evolving technological landscape. As software systems grow more complex, predicting their quality, identifying potential defects, and foreseeing maintenance challenges have become critical tasks. Software Quality Prediction Models (SQPMs) serve as powerful tools to forecast software defects, allowing organizations to take proactive measures in improving system reliability, reducing costs, and enhancing overall product quality. By leveraging Machine Learning (ML) techniques, SQPMs help automate the prediction process, providing decision-makers with valuable insights into which components or modules of software are more prone to defects or changes. This, in turn, enables effective resource allocation, improved maintenance strategies, and a reduction in post-release failures.

The increasing demand for efficient software quality models has led to extensive research on improving the performance of these models. This thesis, titled 'Development and Validation of Software Quality Prediction Models using Machine Learning Techniques', aims to enhance the performance of prediction models by developing robust classifiers and tackling critical issues that hinder the effectiveness of learning algorithms. This research outlines several key objectives, including conducting systematic reviews on the classification algorithms and various factors affecting software quality prediction models. A detailed analysis of the data imbalance problem, parameter tuning, feature selection, and techniques for handling outliers and multi-collinearity are explored. Additionally, the research focuses on the validation of open-source datasets and the proposal of new metrics for software quality assessment. The thesis covers multiple experiments and methodologies aimed at tackling these objectives. Each experiment involves the application of state-of-the-art techniques and algorithms

in software defect categorization (SDC) and software defect prediction (SDP).

The first area of study focuses on the categorization of software defects based on maintenance effort, change impact, and a combination of the two. Various machine learning algorithms, including the Multinomial Naïve Bayes (NBM), ensemble learners (Random Forest (RF), eXtreme Gradient Boosting (XGB), Adaptive Boosting (ADB), and Bagging (BAG)) and Convolutional Neural Networks (CNN) were applied to develop software defect categorization (SDC) models. These models are utilized to categorize software defects into low, medium, and high categories based on three key defect attributes - maintenance effort, change impact, and their combined effect. Experiments were conducted on five Android applications - Bluetooth, Browser, Calendar, Camera, and MMS with the Top10, Top25, Top50, and Top100 relevant keywords extracted from the defect reports through text mining. The results, validated using AUC values, indicate acceptable predictability for defects categorized in various categories. Models based on combined approach demonstrated better performance than those built using only change impact and remain competitive with those built using only maintenance effort. Multinomial Naïve Bayes (NBM) and Convolutional Neural Networks (CNNs) can be effectively used for software defect categorization. Random Forest emerged as the most effective ensemble technique, followed by Bagging and AdaBoost. These models provide valuable insights for software practitioners in terms of effort estimation and resource management, offering a practical solution for addressing high-category defects that demand significant developer or tester effort.

This thesis examines the application of parameter tuning techniques to software quality prediction models. Through a systematic literature review of 31 primary studies, this review identifies and analyzes various parameter tuning methods. A detailed analysis of these studies revealed that tuned models consistently demonstrated improved predictive performance and stability over untuned models. Among the most effective parameter tuning techniques identified were grid search, differential evolution, genetic algorithm-based approaches, and hybrid methods. Classification

algorithms such as Support Vector Machine (SVM), k-Nearest Neighbor (KNN), Random Forest (RF), Neural Networks (NN), and Classification and Regression Trees (CART) were frequently subjected to parameter tuning, with notable sensitivity to hyperparameter adjustments. On the other hand, algorithms like Linear Regression, Regression Trees (RTs), and Bagging with RTs exhibited lower sensitivity to parameter tuning. The results also showed that parameter tuning significantly enhanced the performance of underperforming classifiers, affecting their ranking. The thesis offers practical recommendations for software practitioners, advocating for the use of tuning methods in predictive modeling to ensure optimal performance.

A major challenge in predictive modeling is handling imbalanced datasets in software defect prediction (SDP). This thesis explores two approaches to address this challenge with focus on neural networks: (1) Data-Level Approach: This approach employed four oversampling techniques (ROS, SMOTE, BL-SMOTE, ADASYN), six undersampling techniques (RUS, CC, NM, CNN, TL, ENN), and two hybrid techniques combining oversampling and undersampling (SMOTE-TL, SMOTE-ENN) applied across six open-source software datasets. A total of 78 ANN-based defect prediction models were developed using these techniques, and stratified 10-fold cross-validation was performed to ensure robust validation. The performance was assessed using AUC, G-Mean, and Balance metrics. The study found that oversampling and hybrid techniques, especially SMOTE-ENN, significantly enhanced model performance. Statistical analysis further confirmed that SMOTE-ENN, BL-SMOTE, SMOTE-TL, ROS, and SMOTE were the best-performing techniques, whereas models without resampling and those using undersampling ranked poorly. (2) Algorithm-Level Approach: A Weighted Loss Function for Neural Networks (WL-NN) was introduced to address data imbalance during model training. Four models were developed: NN on imbalanced data, WL-NN on imbalanced data, NN on balanced data via SMOTE-ENN (NN + SMOTE-ENN), and WL-NN on balanced data via SMOTE-ENN (WL-NN + SMOTE-ENN), resulting in 88 defect prediction models. The results demonstrated

that WL-NN significantly improved model performance, with WL-NN + SMOTE-ENN showing a 27% improvement compared to other models. This combination outperformed all approaches, establishing that weighted loss functions combined with data resampling are highly effective in addressing imbalanced data issues in SDP.

Feature selection plays a vital role in improving the performance of SQPMs by eliminating irrelevant or redundant features from the dataset. This thesis focused on evaluating the effectiveness of swarm intelligence techniques - Ant Colony Optimization (ACO), Cuckoo Search (CS), and Crow Search (CRS) - for feature selection in software defect prediction, comparing them to traditional filter-based methods such as Chi-Square (CHI2) and Information Gain (IG). Using 22 datasets from the AEEEM, JIRA, and PROMISE repositories, the results demonstrated that Cuckoo Search (CS) consistently outperformed the other methods, achieving the highest AUC values across most datasets and classifiers. Crow Search (CRS) also performed well, often ranking just behind CS, especially when combined with classifiers like Support Vector Machine (SVM) and Random Forest (RF). In contrast, Ant Colony Optimization (ACO) showed mixed results, delivering strong performance in some cases but lacking consistency compared to CS and CRS. Overall, the study highlights the superior performance of swarm intelligence techniques, with Cuckoo Search (CS) and Crow Search (CRS) emerging as promising approaches to enhance defect prediction models, especially when integrated with advanced classifiers like RF and SVM.

This thesis provides a comprehensive review of multi-collinearity in software quality prediction, a critical issue affecting the reliability, maintainability, and efficiency of predictive models. Through a detailed analysis of the literature, the paper highlights several challenges posed by multi-collinearity, such as uncertainty in predictor effects, overfitting, and reduced generalizability of software quality models. To address these challenges, the review explores various mitigation strategies, including traditional methods like principal component analysis (PCA), regularization techniques (ridge and lasso regression), stepwise regression, and variance inflation factor (VIF) thresh-

olding. It also discusses more recent advancements such as hybrid PCA-regularization approaches, sparse partial least squares (SPLS), and modern machine learning techniques, including Ensemble Learning and Deep Learning approaches. Among these, PCA, ridge regression, and lasso regression are identified as the most commonly employed techniques to combat multi-collinearity.

This thesis proposes a comprehensive metric suite tailored for evaluating event-driven software systems, which are increasingly prevalent due to their capacity to manage complex and asynchronous interactions. The study emphasizes the distinctiveness of event-driven systems compared to structured and object-oriented paradigms, underscoring the need for specialized metrics. The proposed metrics are categorized into key areas, including event structure, event dependency, event performance, event complexity, event synchronization, and event reliability. Each metric is thoroughly defined to provide a standardized, objective framework for assessing the unique characteristics and behavior of event-driven systems.

In summary, this thesis contributes significantly to the field of software quality predictive modeling by addressing key issues such as imbalanced data, parameter tuning, feature selection, and algorithm optimization. The developed models and techniques have demonstrated their potential to improve the accuracy and efficiency of predictive models, thereby supporting better decision-making in software maintenance and quality assurance.

Contents

List of Tables

List of Figures

List of Publications

Papers Accepted/Published in International Journals

1. Ruchika Malhotra and Madhukar Cherukuri, "A Systematic Review of Hyperparameter Tuning Techniques for Software Quality Prediction Models", *Intelligent Data Analysis*, 2024. https://doi.org/10.3233/ida-230653.

2. Ruchika Malhotra and Madhukar Cherukuri, "Convolutional Neural Networks for Software Defect Categorization: An Empirical Validation", *Journal of Universal Computer Science*, 2024.

3. Ruchika Malhotra and Madhukar Cherukuri, "Multi-Collinearity in Software Quality Prediction: Review of Challenges and Solutions", *International Journal of Management, Technology and Engineering*, 2024. https://doi.org/16.10089.IJMTE.2024.V14I8.24.524037.

Papers Accepted/Published in International Conferences

4. Ruchika Malhotra and Madhukar Cherukuri, "Software Defect Categorization based on Maintenance Effort and Change Impact using Multinomial Naïve Bayes Algorithm, *In IEEE - 2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pp. 1068-1073, Amity University, Noida, India, 2020. https://doi.org/10.1109/ICRITO48877.2020.9198037.

5. Ruchika Malhotra and Madhukar Cherukuri, "Metric Suite for Event-Driven Software Systems", *In IEEE - 2023 International Conference on Advances in Computation, Communication and Information Technology (ICAICCIT)* , pp. 1140-1145, Faridabad, India, 2023. https://doi.org/10.1109/ICAICCIT60255.2023.10466151.

Papers Communicated in International Journals

6. Ruchika Malhotra and Madhukar Cherukuri, "An Empirical Validation of Convolutional Neural Networks for Software Bug Prediction Models", *Multimedia Tools and Applications*. (**Under Revision**).

7. Ruchika Malhotra and Madhukar Cherukuri, "Towards Optimal Software Defect Categorization: A Multi-Aspect Analysis of Maintenance Effort, Change Impact, and Combined Factors via Random Forest, Naïve Bayes, and CNN", *Intelligent Data Analysis*. (**Under Revision**).

8. Ruchika Malhotra and Madhukar Cherukuri, "Imbalanced Data Learning in Defect Prediction: A Weighted Loss Function Approach for Neural Networks", *Advances in Data Analysis and Classification*. (**Peer Review**).

9. Ruchika Malhotra and Madhukar Cherukuri, "Application of Imbalanced Learning Techniques on Artificial Neural Network for Software Defect Prediction", *Journal of Intelligent & Fuzzy Systems*. (**Peer Review**).

10. Ruchika Malhotra and Madhukar Cherukuri, "Swarm Intelligence-Based Feature Selection for Software Defect Prediction", *Kuwait Journal of Science*. (**Communicated**).

11. Ruchika Malhotra and Madhukar Cherukuri, "Deep Learning for Software Defect Categorization", *Applied Soft Computing*. (**Communicated**).

List of Abbreviations

AAE	**Average Absolute Error**
ACM	**Association for Computing Machinery**
ACO	**Ant Colony Optimization**
ADASYN	**Adaptive Synthetic Sampling**
ADB	**Adaptive Boosting**
AHF	**Attribute Hiding Factor**
AHS	**Automatic Hybrid Search**
AI	**Artificial Intelligence**
AIF	**Attribute Inheritance Factor**
AMC	**Average Method Complexity**
ANA	**Average Number of Ancestors**
ANOVA	**Analysis of Variance**
API	**Application Programming Interface**
ARE	**Average Relative Error**
ARFF	**Attribute-Relation File Format**
AUC	**Area under the ROC curve**
AUCPR	**Area under Precision-Recall Curve**
BAG	**Bagging**
BL-SMOTE	**Borderline Synthetic Minority Oversampling Technique**
Ca	**Afferent Coupling**
CAM	**Cohesion among Methods of a Class**
CART	**Classification and Regression Trees**
CBM	**Coupling Between Methods of a Class**

CBO	**Coupling Between Object**
CC	**Cluster Centroids**
Ce	**Efferent Coupling**
CF	**Coupling Factor**
CFS	**Correlationbased Feature Selection**
CHI2	**Chi-Square**
CI	**Class Interface**
CIS	**Class Interface Size**
CNN	**Convolutional Neural Network**
COF	**Coupling Factor**
COH	**Cohesion**
CP	**Correlation Percentile**
CRS	**Crow Search**
CS	**Cuckoo Search**
CSV	**Comma-Separated Values**
CYC	**Cyclomatic Complexity**
DAC	**Data Abstraction Coupling**
DAM	**Data Access Metric**
DBN	**Deep Belief Network**
DCC	**Direct Class Coupling**
DCRS	**Defect Collection and Reporting System**
DE	**Differential Evolution**
DIT	**Depth of Inheritance Tree**
DNN	**Deep Neural Network**
DSBF	**Domain-Specific Bias Focusing**
DSC	**Design Size in Classes**
DTB	**Deep Transfer Biclustering**
EDA	**Exploratory Data Analysis**
EDA	**Event-Driven Architectures**
EDP	**Event-Driven Programming**

ENN	**Edited Nearest Neighbor Rule**
ES	**Exhaustive Search**
FDA	**Flexible Discriminant Analysis**
FN	**False Negative**
FP	**False Positive**
FPR	**False Positive Rate**
GA	**Genetic Algorithm**
GAMBoost	**Generalized linear and Additive Models Boosting**
GBM	**Gradient Boosting Machine**
GNN	**Graph-based Neural Networks**
GPLS	**Generalized Partial Least Squares**
HS	**Heuristic Search**
Ibk	**Instance-based learning with parameter k**
IC	**Inheritance Coupling**
ICH	**Information flow based Cohesion**
ICP	**Information flow-based coupling**
IEEE	**Institute of Electrical and Electronics Engineers**
IG	**Information Gain**
IH-ICP	**Inheritance-based Coupling**
ISO	**International Organization for Standardization**
KM-SMOTE	**K-Means Synthetic Minority Oversampling Technique**
KNN	**k-Nearest Neighbor**
KS	**KolmogorovSmirnov Class Correlation-Based Filter**
LCOM	**Lack of Cohesion in Methods**
LDA	**Linear Discriminate Analysis**
LIME	**Local Interpretable Modelagnostic Explanations**
LMT	**Logistic Model Trees**
LOC	**Lines of Code**
LOO-CV	**Leave-One-Out Cross-Validation**

LR	**Linear Regression**
LSTM	**Long Short Term Memory**
MAE	**Mean Absolute Error**
MCC	**Matthews's correlation coefficient**
MDP	**Metrics Data Program**
MDS	**Multidimensional Scaling**
MFA	**Method of Functional Abstraction**
MHF	**Method Hiding Factor**
MIF	**Method Inheritance Factor**
ML	**Machine Learning**
MLP	**Multilayer Perceptron**
MOA	**Measure of Aggression**
MOOD	**Metrics for Object Oriented Design**
MPC	**Message Pass Coupling**
MPC	**Methods per Class**
NASA	**National Aeronautics and Space Administration**
NB	**Naïve Bayes**
NBM	**Multinomial Naïve Bayes**
NECM	**Normalized Expected Cost of Misclassification**
NIH-ICP	**Non-Inheritance-based Coupling**
NLP	**Natural Language Processing**
NM	**Near Miss**
NN	**Neural Network**
NOA	**Number of Operations Added by a subclass**
NOC	**Number Of Children**
NOH	**Number of Hierarchies**
NOM	**Number of Methods**
NOMA	**Number of Object/Memory Allocation**
NOO	**Number of Operations (methods) Overridden by a subclass**
NOP	**Number of Polymorphic Methods**

NPM	**Number of Public Methods**
NSF	**Number of Static Fields**
NSGA-II	**Non-dominated Sorting Genetic Algorithm II**
NSM	**Number of Static Methods**
OO	**Object Oriented**
PCA	**Principal Component Analysis**
PD	**Probability of Detection**
PDA	**Penalized Discriminant Analysis**
PF	**Polymorphism Factor**
PF	**Probability of False Alarm**
POF	**Polymorphism Factor**
PSO	**Particle Swarm Optimization**
QMOOD	**Quality Model for Object-Oriented Design**
ReLU	**Rectified Linear Unit**
RF	**Random Forest**
RFC	**Response For a Class**
RFE	**Recursive Feature Elimination**
RMSE	**Root Mean Square Error**
RNN	**Recurrent Neural Networks**
ROC	**Receiver Operating Characteristic**
ROS	**Random Over Sampling**
RR	**Ridge Regression**
RS	**Random search**
RT	**Regression Trees**
RUS	**Random Under Sampling**
SA	**Simulated Annealing**
SBC	**Software Bug Categorization**
SDC	**Software Defect Categorization**
SDP	**Software Defect Prediction**
SHAP	**SHapley Additive exPlanations**

SIX	**Specialization Index**
SLG	**Square Loss Gradient Boost**
SMOTE	**Synthetic Minority Oversampling Technique**
SMOTE-ENN	**SMOTE with Edited Nearest Neighbors**
SMOTE-TL	**SMOTE with Tomek Links**
SMP	**Software Maintainability Prediction**
SPLS	**Sparse Partial Least Squares**
SQPM	**Software Quality Prediction Model**
SSA	**Sparrow Search Algorithm**
SU	**Symmetrical Uncertainty**
SVM	**Support Vector Machine**
TCA	**Transfer Component Analysis**
TL	**Tomek Links**
TN	**True Negative**
TNR	**True Negative Rate**
TP	**True Positive**
TPE	**Tree-of-Parzen Estimators**
TPR	**True Positive Rate**
VIF	**Variance Inflation Factor**
VT	**Variance Threshold**
WL-NN	**Weighted Loss Function for Neural Networks**
WMC	**Weighted Methods per Class**
XGB	**eXtreme Gradient Boosting**
xGBTree	**eXtreme Gradient Boosting Tree**

Chapter 1

Introduction

1.1 Introduction

In an era where software systems permeate every facet of modern society, ensuring the quality of these systems is paramount. The pervasive reliance on software in critical domains, such as healthcare, finance, defense, and infrastructure, amplifies the consequences of software failures, which can range from financial losses to loss of life. Consequently, software quality has evolved into a critical area of study within software engineering, attracting considerable attention from both academia and industry [1].

The development and maintenance of software systems are inherently complex, often requiring the integration of numerous components, technologies, and processes. As software systems become more intricate, the potential for defects and quality issues increases, posing significant challenges to software engineers and quality assurance teams [2]. Traditional approaches to ensuring software quality, such as manual testing and code

reviews, are becoming increasingly inadequate in the face of the growing complexity and scale of modern software systems. These approaches are often time-consuming, labor-intensive, and prone to human error, making them insufficient to meet the rigorous demands of contemporary software development.

In this context, predictive modeling has emerged as a powerful tool for enhancing software quality assurance. Predictive modeling leverages historical data to forecast future outcomes, enabling software engineers to identify and address potential quality issues before they manifest in operational environments. By applying machine learning (ML) techniques to software quality data, predictive models can provide valuable insights into defect-prone components, guide testing efforts, and inform decision-making processes [3].

The central premise of this work is that the performance of predictive models in software quality assurance can be significantly improved by addressing several key factors that affect model accuracy, reliability, and generalizability. These factors include imbalanced data, outliers, overfitting and underfitting, multi-collinearity, parameter tuning, and feature selection [4]. By systematically investigating these factors and developing novel approaches to mitigate their impact, this research aims to advance the state of the art in software quality prediction and contribute to the broader field of software engineering.

This chapter provides a structured foundation for the thesis by introducing key concepts and setting the context for the research. It begins with an overview of software quality in Section 1.2. Section 1.3 explains predictive modeling, breaking down its definition, steps, and application in software quality prediction, followed by an exploration of techniques used in this field. Section 1.4 discusses the various factors that influence the performance of predictive models. A literature survey in Section 1.5 reviews existing studies on software metrics and software quality prediction, identifying research gaps and directions. The

chapter continues by outlining the thesis objectives in Section 1.6, which defines the vision, focus, and goals of the research. Section 1.7 provides a summary of the research work, and the chapter concludes with a brief overview of the thesis organization in Section 1.8.

1.2 Software Quality

A software system is designed and developed to meet a set of well-defined requirements (functional and non-functional). Software quality is defined as the extent to which a software system meets the specified requirements. Software quality is a multifaceted concept that encompasses various attributes of a software product, including its functionality, performance, reliability, security, usability, and maintainability. The International Organization for Standardization (ISO) defines software quality as *"the degree to which a software product meets specified requirements and user needs."* According to IEEE Standard glossary of Software Engineering Terminology, Software Quality is defined as [5]: *"1. The degree to which a system, component, or process meets specific requirements and 2. The degree to which a system, component, or process meets customer or user needs or expectations."* These definitions highlights the dual focus of software quality on both conformance to requirements and the satisfaction of user expectations.

The impact of software quality on an organization can be profound, influencing everything from operational efficiency to customer satisfaction. In the context of commercial software, quality directly affects the marketability and competitiveness of a product. A software product that is prone to defects, crashes, or security vulnerabilities is likely to face poor adoption rates, negative user reviews, and increased support costs [6]. Conversely, high-quality software that performs reliably and meets user expectations is more likely to

succeed in the marketplace. In the context of safety-critical systems, such as those used in healthcare, aviation, or automotive industries, software quality takes on an even greater significance. High-quality software is essential for ensuring that these systems function correctly, efficiently, and securely. Conversely, poor software quality can lead to a wide range of negative consequences, including software failures, security breaches, and user dissatisfaction. In extreme cases, software defects can result in catastrophic failures with significant financial, legal, and reputational repercussions [7]. In these domains, software failures can have life-threatening consequences. For example, a defect in medical device software could lead to incorrect diagnoses or treatment recommendations, potentially endangering patients' lives. Similarly, software failures in automotive systems could lead to accidents and fatalities. As a result, ensuring the highest possible level of software quality is imperative in these contexts.

Ensuring software quality is a complex and challenging task, particularly in the context of modern software systems [8]. Several factors contribute to this complexity, including the growing scale and intricacy of software systems, the diversity of technologies and platforms, and the dynamic nature of software requirements. Additionally, the prevalence of distributed and agile development practices, which involve multiple teams working on different components of a software system, further complicates the task of ensuring consistent quality across the entire system. A key challenge in maintaining software quality is the difficulty of detecting and preventing defects during the software development process. Despite the availability of various testing and verification techniques, it is often impossible to identify all potential defects before a software product is released. This is particularly true for extensive and intricate systems, where the number of possible test cases and execution paths can be virtually infinite. As a result, even well-tested software

products may contain defects that only become apparent after deployment. *In this work, the terms "defect","bug" and "fault" are used interchangeably.*

1.3 Prediction Modeling

This section discusses predictive modeling, the validation process, the steps for SQPM.

1.3.1 What is Predictive Modeling?

Predictive modeling is a statistical and computational technique used to forecast future events or outcomes based on historical data [9]. It is widely used in various domains, including finance, healthcare, marketing, and engineering, to predict outcomes such as stock prices, disease outbreaks, customer behavior, and equipment failures. In the context of software quality, predictive modeling is used to forecast the likelihood of defects, categorize issues, and guide quality assurance activities. Predictive modeling is inherently interdisciplinary, drawing on concepts and techniques from statistics, machine learning, data mining, and domain-specific knowledge. It involves the use of mathematical models to represent the relationships between input variables (features) and the outcome of interest (target variable). These models are trained on historical data, allowing them to learn patterns and relationships that can be used to make predictions on new, unseen data.

1.3.2 Steps in Prediction Modeling

The following are the steps that are used in Predicting modeling:

1. **Defining the Objective:** The first and most critical step in predictive modeling is to

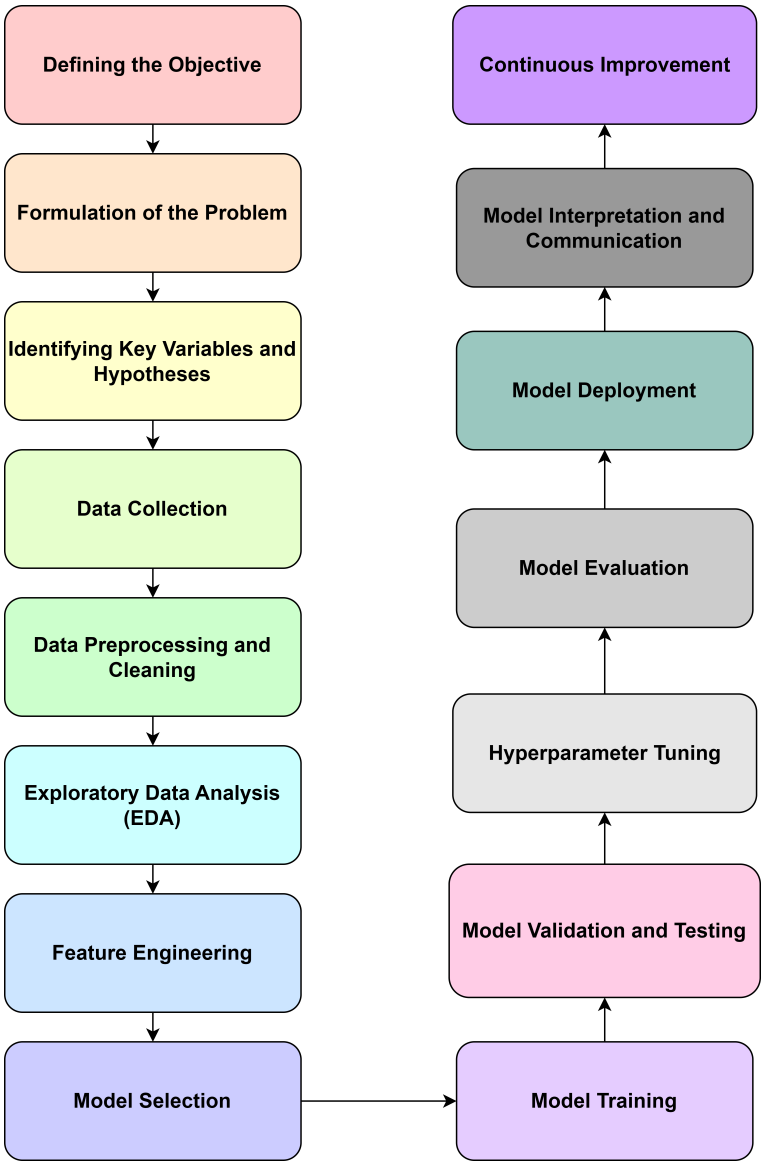

Figure 1.1: Steps in Predictive Modeling

clearly define the objective of the modeling effort. This involves understanding what you are trying to predict, why it is important, and how the predictions will be used.

In the context of software quality, the objective might be to predict the likelihood of defects in a software module, the time required for a bug to be resolved, or the impact of a change on software maintainability. Defining a clear objective helps in guiding the subsequent steps of the modeling process and ensures that the model's output will be relevant and actionable. The objective should be specific, measurable, achievable, relevant, and time-bound (SMART) [10].

2. **Formulation of the Problem:** Once the objective is defined, the next step is to formulate the problem in a way that can be addressed using predictive modeling techniques. This involves translating the business or research objective into a formal problem statement that defines the target variable (what you want to predict) and the independent variables (features) that will be used to make the prediction. For example, if the objective is to predict software defects, the problem formulation might involve defining the target variable as a binary outcome (defect or no defect) and identifying potential predictors such as code complexity, lines of code, developer experience, and past defect history. Problem formulation also involves deciding on the type of predictive model that is appropriate for the task. This could be a classification model (e.g., predicting whether a software module will have defects), a regression model (e.g., predicting the number of defects), or a ranking model (e.g., prioritizing software modules based on their defect risk).

3. **Identifying Key Variables and Hypotheses:** In this step, the key variables that are likely to influence the target outcome are identified. These variables are usually selected based on domain knowledge, previous research, or exploratory data analysis. Along with identifying key variables, it is also essential to formulate hypotheses

regarding the relationships between these variables and the target outcome. For example, in software quality prediction, one might hypothesize that more complex code (as measured by metrics such as cyclomatic complexity) is more likely to contain defects. These hypotheses guide the selection of features and the design of the model.

4. **Data Collection:** After the problem is formulated and key variables are identified, the next step is to collect the data that will be used to build and train the predictive model. Data collection involves gathering relevant data from various sources, such as software repositories, version control systems, bug tracking systems, and other software engineering tools. The quality and relevance of the data are critical to the success of the predictive model. Data collection should focus on acquiring data that is complete, accurate, and representative of the problem at hand. In some cases, data from multiple sources may need to be integrated to provide a comprehensive view of the software system.

5. **Data Preprocessing and Cleaning** Once the data is collected, it must be preprocessed and cleaned to ensure it is suitable for modeling. This step involves several tasks, including handling missing values, correcting errors, normalizing data, and transforming variables if necessary. In software quality prediction, data preprocessing might also involve aggregating data at the appropriate level (e.g., at the module or file level) and creating new features that capture relevant aspects of the software development process. Outlier detection and handling are also important at this stage. Outliers can skew the results of the predictive model, leading to biased or inaccurate predictions. Techniques such as z-score analysis, IQR-based methods, or

domain-specific rules are often used to identify and address outliers.

6. **Exploratory Data Analysis (EDA):** Exploratory Data Analysis (EDA) is an essential step in predictive modeling that involves analyzing the data to discover patterns, trends, and relationships between variables. EDA helps in understanding the distribution of data, identifying any anomalies, and determining the suitability of various features for predictive modeling. In EDA, various statistical and visualization techniques are employed to summarize the data and generate insights. For example, in software quality prediction, one might use histograms, box plots, and scatter plots to examine the distribution of software metrics and their relationship with defect occurrence. EDA also plays a critical role in feature selection, helping to identify the most relevant features that should be included in the model. This step is crucial for building models that are both accurate and interpretable.

7. **Feature Engineering:** Feature engineering is the process of transforming raw data into features that better represent the underlying problem to the predictive model. This step is critical because the quality of the features can significantly impact the performance of the model. In software quality prediction, feature engineering might involve creating new features that capture aspects of software complexity, change history, or developer activity. For example, one might create a feature that represents the number of changes a file has undergone in the last month or a feature that captures the experience level of the developers who contributed to a particular module. Feature engineering also includes the selection of the most relevant features (feature selection) and the transformation of features to improve their suitability for modeling (e.g., by applying logarithmic transformations to handle skewed data).

8. **Model Selection:** With the data preprocessed and the features engineered, the next step is to select the appropriate predictive model. The choice of model depends on the nature of the problem, the type of data available, and the specific requirements of the prediction task. In software quality prediction, common models include decision trees, random forests, support vector machines, neural networks, and logistic regression. The selection process often involves comparing the performance of different models using cross-validation or other techniques to determine which model best meets the objective of the prediction task.

9. **Model Training:** Once the model is selected, it is trained using the prepared data. Model training involves fitting the model to the data, allowing it to learn the patterns and relationships between the input features and the target variable. During training, the model's parameters are adjusted to minimize the error between the predicted and actual outcomes. This step is iterative and may involve several rounds of training and validation to optimize the model's performance.

10. **Model Validation and Testing:** Validation and testing are essential steps that ensure the model's performance is generalizable to new, unseen data. Validation typically involves partitioning the data into training and validation sets, where the model is trained on one subset of the data and validated on another. Cross-validation techniques, such as k-fold cross-validation, are often used to assess the model's robustness and to avoid overfitting. Once the model performs well on the validation set, it is tested on a separate test set to evaluate its effectiveness in a real-world applications.

11. **Hyperparameter Tuning:** It is the process of optimizing the model's hyperparame-

ters to achieve the best possible performance. Hyperparameters are the parameters that are not learned during training but are set before the training process begins. These include parameters such as the learning rate, the number of trees in a random forest, or the depth of a neural network. Various techniques, such as grid search, random search, or Bayesian optimization, can be used to systematically explore the hyperparameter space and find the optimal settings.

12. **Model Evaluation:** After the model is trained and tested, it is evaluated using appropriate metrics to assess its performance. Common evaluation metrics for predictive models include accuracy, precision, recall, F1-score, and area under the ROC curve (AUC-ROC). In the context of SQP, it is also important to consider the practical implications of the model's performance, such as the cost of false positives and false negatives. For example, predicting a defect that does not exist (false positive) may lead to unnecessary effort, while failing to predict an existing defect (false negative) may result in undetected issues in the software.

13. **Model Deployment:** Once the model is validated and evaluated, it is deployed in a production environment where it can be used to make predictions on new data. Model deployment involves integrating the predictive model into existing systems, such as software development tools or quality assurance platforms, so that it can be used in real-time or batch processing. Deployment also requires ongoing monitoring and maintenance to ensure that the model continues to perform well over time. This includes tracking the model's predictions, retraining the model as new data becomes available, and updating it to reflect changes in the underlying software development process.

14. **Model Interpretation and Communication:** Model interpretation involves explaining the predictions made by the model in a way that is understandable and actionable for stakeholders. This is particularly important in software quality prediction, where decisions based on model predictions can have significant implications for software development and maintenance. Techniques such as feature importance analysis, partial dependence plots, and SHAP (SHapley Additive exPlanations) values can be used to interpret the model's predictions and understand the contribution of different features. Communication of the model's results is also critical, involving the presentation of findings to stakeholders in a clear and concise manner. This may include reports, dashboards, or visualizations that highlight the key insights and recommendations based on the model's predictions.

15. **Continuous Improvement:** Predictive modeling is an ongoing process that requires continuous improvement. As new data becomes available and as the software development environment evolves, the predictive model may need to be updated or retrained to maintain its accuracy and relevance. Continuous improvement involves regularly reviewing the model's performance, incorporating feedback from stakeholders, and making adjustments as necessary. This iterative approach ensures that the model remains effective in predicting software quality and supporting decision-making in software development.

1.3.3 Predictive Modeling for Software Quality

Predictive modeling has become an integral part of software quality assurance, providing a data-driven approach to identifying and mitigating quality issues. The application of

predictive modeling in software quality spans several key areas, including defect prediction, defect categorization, change prediction, maintainability prediction and effort estimation.

1.3.3.1 Defect Prediction

Defect prediction is one of the most widely studied applications of predictive modeling in software quality. The goal of defect prediction is to identify components or modules in a software system that are likely to contain defects [8]. By identifying defect-prone components early in the development phase, software engineers can prioritize testing and quality assurance efforts, thereby reducing the risk of defects in the final product. Defect prediction models typically use historical data, such as software metrics (e.g., lines of code, complexity, code churn) and defect reports, to forecast the likelihood of defects in new or modified components. These models may be based on a variety of machine learning techniques, including decision trees, random forests, support vector machines, and neural networks. The choice of model and features is critical to the success of defect prediction, as it determines the model's ability to accurately identify defect-prone components.

1.3.3.2 Defect Categorization

Defect categorization involves classifying defects based on their severity, impact, or other attributes [11]. This is crucial for prioritizing quality assurance efforts, as different types of defects may require different levels of attention and resources. For example, a critical defect that affects the core functionality of the software may need to be addressed immediately, while a minor defect that affects a rarely used feature may be deferred to a later release. Predictive models for defect categorization typically use features such as defect descriptions, defect history, and software metrics to classify defects into different

categories. These models may be based on classification algorithms such as multinomial Naïve Bayes, support vector machines, or convolutional neural networks. The accuracy of defect categorization models is critical for ensuring that resources are allocated effectively and that the most critical defects are addressed promptly.

1.3.3.3 Change Prediction

Change prediction models aim to forecast the likelihood of future changes in specific components of a software system [12]. Changes may be required due to new feature requests, bug fixes, or adaptations to new environments. Predicting which parts of the software are more susceptible to change can help in better planning and resource management, allowing for proactive maintenance and reducing the risk of introducing new defects. Change prediction models often utilize version control data, change history, and dependency graphs to forecast which modules are likely to undergo changes in future iterations.

1.3.3.4 Maintainability Prediction

Software maintainability is a critical quality attribute that determines how easily a software system can be modified to correct defects, improve performance, or adapt to a changed environment [13]. Maintainability prediction models assess the ease with which software can be maintained based on different factors such as code structure, documentation quality, and adherence to coding standards. These models help in identifying areas of the software that may become maintenance bottlenecks, enabling organizations to take corrective actions early in the development process. Predictive models for maintainability often rely on metrics like code complexity, cohesion, coupling, and code smells.

1.3.3.5 Reliability Prediction

Reliability prediction models are developed to estimate the probability of software functioning without failure over a specified period under given conditions [14]. These models are particularly important for critical systems where failures can have severe consequences. Reliability prediction often involves analyzing historical failure data, operational profiles, and software architecture. The goal is to forecast the reliability of future releases or new features, allowing organizations to allocate resources to areas that require the most attention to ensure high reliability.

1.3.3.6 Effort Estimation

Effort estimation involves predicting the amount of effort required to complete a software development task, such as fixing a defect, implementing a feature, or conducting a code review [15]. Accurate effort estimation is critical for project planning, resource allocation, and budget management. Overestimating effort can lead to inefficient use of resources, while underestimating effort can result in missed deadlines, cost overruns, and compromised quality. Predictive models for effort estimation typically use features such as task complexity, developer experience, and historical effort data to forecast the effort required for new tasks. These models may be based on regression algorithms, such as linear regression, decision trees, or neural networks. The accuracy of effort estimation models is critical for ensuring that projects are completed on time, within budget, and with the desired level of quality.

1.3.4 Techniques Applied in Predictive Modeling for Software Quality

Machine learning (ML) techniques form the backbone of predictive modeling in software quality assurance. These techniques enable the development of models that can learn patterns from historical data and make accurate predictions on new data. Several ML techniques are commonly applied in predictive modeling for software quality, including supervised learning, unsupervised learning, and ensemble learning.

1.3.4.1 Supervised Learning

It is the most widely used ML technique in predictive modeling for software quality [16]. In supervised learning, the model is trained on labeled data, where the input features (e.g., software metrics) are associated with known outcomes (e.g., defect presence). The model learns the mapping between the features and the outcomes, allowing it to make predictions on new, unlabeled data. Common supervised learning algorithms used in software quality prediction include decision trees, random forests, support vector machines, and neural networks. These algorithms are particularly effective for tasks such as defect prediction, defect categorization, and effort estimation, where the goal is to predict a specific outcome based on historical data.

1.3.4.2 Unsupervised Learning

it is another important ML technique used in predictive modeling, particularly for tasks such as anomaly detection and clustering. In unsupervised learning, the model is trained on unlabeled data, where the input features are not associated with specific outcomes [17]. The model learns the underlying structure of the data, such as clusters or patterns, without

the need for labeled data. In the context of software quality, unsupervised learning can be used for tasks such as identifying outliers in software metrics, detecting unusual patterns in code changes, or clustering similar defects based on their characteristics. Common unsupervised learning algorithms include k-means clustering, hierarchical clustering, and principal component analysis (PCA).

1.3.4.3 Ensemble Learning

Ensemble learning involves combining multiple ML models to improve predictive performance. The idea behind ensemble learning is that by aggregating the predictions of multiple models, the overall prediction can be more accurate and robust than any individual model [18]. This is particularly useful in software quality prediction, where different models may capture different aspects of the data. Common ensemble learning techniques include bagging (e.g., random forests), boosting (e.g., gradient boosting machines), and stacking. These techniques have proven to enhance the accuracy and robustness of predictive models, particularly in complex and high-dimensional datasets.

1.3.4.4 Deep Learning

It is a subset of ML that involves the use of neural networks with multiple layers (i.e., deep neural networks) to learn complex patterns and representations from data. Deep learning has gained significant attention in recent years due to its success in various domains, such as computer vision, speech recognition, and natural language processing. In the context of software quality, deep learning has shown promise in tasks such as defect prediction, defect categorization, and effort estimation [19]. For example, convolutional neural networks (CNNs) have been applied to the categorization of software defects based

on their descriptions, while recurrent neural networks (RNNs) have been used to model the temporal aspects of software development processes. One of the key advantages of deep learning is its ability to automatically learn features from raw data, reducing the need for manual feature engineering. This is particularly useful in software quality prediction, where the complexity and diversity of the data can make feature engineering challenging. However, deep learning models also demand significant volumes of data and computational resources, which can be a barrier to their adoption in some contexts.

1.3.4.5 Natural Language Processing (NLP)

Natural language processing (NLP) is another emerging trend in predictive modeling for software quality. NLP techniques are used to analyze and interpret human language, making them highly suitable for tasks such as processing defect descriptions, code comments, and other textual data [20]. In the context of software quality, NLP has been utilized to tasks such as defect categorization, sentiment analysis of developer comments, and automatic generation of test cases from requirements. For example, NLP techniques such as word embeddings and transformers have been used to represent textual data in a form that can be used by predictive models. The integration of NLP with traditional ML techniques has the potential to greatly improve the accuracy and relevance of predictive models, particularly in tasks that involve textual data. However, NLP also introduces additional challenges, such as the need to handle ambiguity, context, and domain-specific language.

1.3.4.6 Transfer Learning and Cross-Project Validation

Transfer learning is an ML technique that involves applying knowledge learned from one domain or project to another. In the context of software quality prediction, transfer learning has been used to apply models trained on one project to other projects, particularly in situations where labeled data is scarce. Cross-project validation is a related concept that involves evaluating the performance of a predictive model on data from different projects. This is important for ensuring that the model is generalizable and can be utilized across a broad range of software systems. Cross-project validation is particularly challenging in software quality prediction, as different projects may have different characteristics, metrics, and processes. The application of transfer learning and cross-project validation is an emerging trend in predictive modeling for software quality, with the potential to enhance the generalizability and applicability of predictive models across different contexts.

1.4 Factors Affecting the Performance of Predictive Modeling for Software Quality

The performance of predictive models in software quality prediction is influenced by several factors, each of which can greatly influence the accuracy, reliability, and generalizability of the models. Understanding these factors is critical for developing effective predictive models and improving their performance in practice.

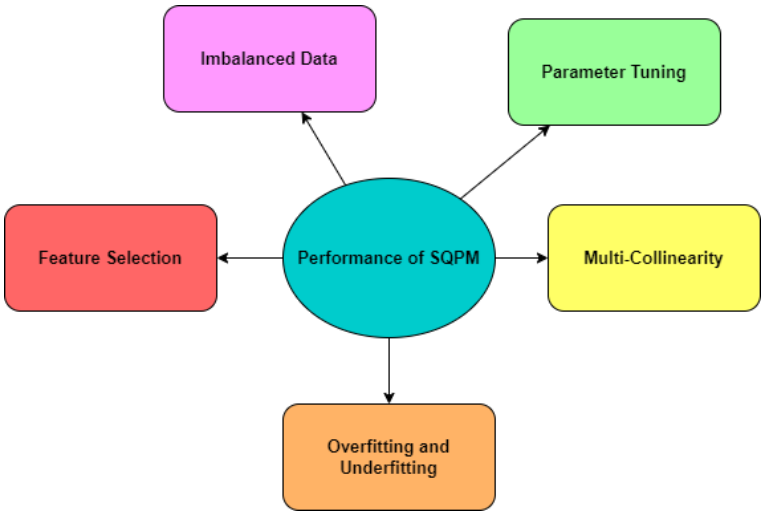

Figure 1.2: Factors affecting the Performance of SQPM

1.4.1 Imbalanced Data

It is a common challenge in predictive modeling for software quality, particularly in tasks such as defect prediction and defect categorization. Imbalanced data occurs when the target variable's distribution is skewed, with one class being significantly more prevalent than the other(s). For example, in defect prediction, the majority of components may be defect-free, while only a small proportion may contain defects. Imbalanced data can lead to biased models that are overly focused on the majority class, resulting in poor performance on the minority class. This is particularly problematic in software quality prediction, where the minority class (e.g., defective components) is often the most critical to identify. Several techniques have been developed to address imbalanced data, including resampling methods (e.g., oversampling, undersampling), cost-sensitive learning, and ensemble methods.

1.4.2 Parameter Tuning

Parameter tuning involves optimizing the hyperparameters of the predictive model to achieve the best possible performance. Hyperparameters are the parameters that are not learned from the data but are set before the training process, such as the learning rate, regularization strength, and the number of hidden layers in a neural network. In the context of software quality prediction, parameter tuning is critical for achieving optimal model performance. Poorly chosen hyperparameters can lead to underfitting, overfitting, or slow convergence, resulting in suboptimal predictive accuracy. Several techniques can be used for parameter tuning, including grid search, random search, and Bayesian optimization.

1.4.3 Feature Selection

It is the process of identifying the most relevant features that contribute to the predictive power of the model. In the context of SQP, feature selection is critical for improving model accuracy, reducing computational complexity, and enhancing interpretability. Irrelevant or redundant features can lead to overfitting, reduced accuracy, and increased computational cost. Several techniques can be used for feature selection, including filter methods, wrapper methods, and embedded methods. The choice of feature selection technique is influenced by the nature of the data, the predictive task, and the specific requirements of the model

1.4.4 Multi-Collinearity

Multi-collinearity occurs when two or more features in the dataset are highly correlated with each other. In the context of software quality, multi-collinearity can occur when

multiple software metrics or process variables capture similar information. For example, lines of code and code complexity may be highly correlated, as larger modules tend to be more complex. Multi-collinearity can lead to unstable model estimates, where minor changes in the data can result in significant changes in the model coefficients. This can make the model difficult to interpret and reduce its predictive accuracy. Several techniques can be used to address multi-collinearity, including feature selection, principal component analysis (PCA), and regularization methods such as ridge regression

1.4.5 Overfitting and Underfitting

Overfitting and underfitting are two common issues that can affect the performance of predictive models. Overfitting happens when the model becomes overly complex and captures noise or random variations in the training data, resulting in poor generalization to new data. Underfitting occurs when the model is too simple and fails to capture the underlying patterns in the data, resulting in poor performance on both the training and test data. In the context of SQP, overfitting and underfitting can lead to inaccurate predictions and unreliable models. Several techniques can be used to mitigate overfitting and underfitting, including regularization, cross-validation, and model complexity control. The choice of model architecture, hyperparameters, and training techniques is critical for achieving the right balance between underfitting and overfitting.

1.5 Literature Survey

Software quality prediction is a vital aspect of software engineering, focusing on forecasting the likelihood of defects, estimating the time required for bug resolution, and

assessing the impact of changes on software maintainability. The increasing complexity of modern software systems and the high demand for reliable software have necessitated the development of sophisticated predictive models that can aid in maintaining high software quality standards. Predictive modeling in software engineering has evolved over the years, incorporating diverse statistical and machine learning techniques to enhance the accuracy and reliability of predictions. This literature review aims to provide a comprehensive overview of the existing research on software quality prediction, highlighting key techniques, challenges, and future directions.

1.5.1 Software Metrics

Some of the frequently utilized Object Oriented metric suites found in the literature include:

- Chidamber and Kemerer[21] proposed a metric suite which consists of 6 metrics pertaining various aspects of OO software. These metrics are: Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number Of Children (NOC), Coupling Between Object (CBO), Response For a Class (RFC) and Lack of Cohesion in Methods (LCOM). This metric suite has been used in various studies predicting software quality prediction.

- Li and Henry[22] had given a metric suite that comprises: Data Abstraction Coupling (DAC), Message Pass Coupling (MPC) and Number of Methods (NOM) and two size metrics namely SIZE1 and SIZE2.

- Lorenz and Kidd[23] presented the following Object-Oriented metrics: Class Size metrics (CS), Number of Operations (methods) Overridden by a subclass (NOO),

Number of Operations Added by a subclass (NOA) and Specialization Index (SIX).

- Metrics for Object Oriented Design (MOOD) [24] contains following metrics: Attribute Hiding Factor (AHF), Method Hiding Factor (MHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Coupling Factor (COF), and Polymorphism Factor (POF).

- Bieman and Kang[25] introduced 2 cohesion metrics: Tight Class Cohesion (TCC) and Loose Class Cohesion (LCC).

- Briand et al.[26] provided 18 metrics that assess different types of interactions between classes. In addition to various coupling metrics proposed by Briand [26], Martin[27] introduced two additional coupling metrics: two more coupling metrics, Afferent Coupling (Ca) and Efferent Coupling (Ce).

- Lee et al.[28] introduced inheritance-based and non-inheritance-based coupling metrics: NIH-ICP, IH-ICP. They also introduced Information flow-based coupling (ICP) metric which is the sum of NIH-ICP and IH-ICP. To assess cohesion, they proposed information flow based cohesion (ICH) metric.

- Bansiya and Davis[29] proposed the Quality Model for Object-Oriented Design (QMOOD) metrics suite. The components of this metric suite are as follows: Number of Polymorphic Methods (NOP), Design Size in Classes (DSC), Number of Hierarchies (NOH), Average Number of Ancestors (ANA), Data Access Metric (DAM), Direct Class Coupling (DCC), Cohesion among Methods of a Class (CAM), Measure of Aggression (MOA), Method of Functional Abstraction (MFA)), Class Interface Size (CIS).

- Tang et al.[30] introduced Coupling Between Methods of a Class (CBM), Number of Object/Memory Allocation (NOMA), Average Method Complexity (AMC), and Inheritance Coupling (IC).

1.5.2 Software Quality Prediction

Malhotra [31] conducted a systematic review of studies from January 1991 to October 2013 in the literature that use the machine learning techniques for software fault prediction and she found that the most frequently used ML techniques for Software Fault Prediction were C4.5, Naive Bayes, Multilayer Perceptron, Support Vector Machines and Random Forest.

The study of Mahmood et. al. [32] showed that when the data is imbalanced, the predictive capability of SDP studies tends to be low. Japkowicz [33] demonstrated that class imbalances hinder the performance of standard classifiers and evaluated the effectiveness of Over-Sampling, Down-Sizing and Learning by Recognition strategies to address the issue. Hulse et al. [34] presented a comprehensive and systematic experimental analysis of learning from imbalanced data, using 11 learning algorithms with 35 real-world benchmark datasets from a variety of application domains. He and Garcia [35] offer an extensive review of the development of research in learning from imbalanced data. Khoshgoftaar and Gao [36] used the random under sampling technique (RUS) on the majority class to alleviate the detrimental effects of imbalanced data on the prediction models. Ozturk and Zengin [37] proposed HSDD (hybrid sampling for defect data sets) to solve imbalanced data problem. Kumar and Sureka [38] applied five different strategies (Random Undersampling, Random Oversampling, SMOTE, SMOTEBoost and RUSBoost) to counter data

imbalance problem and concluded that Gain Ratio and RELEIF outperform other strategies. Kanimozhi [39] proposed that combining different expressions of the re-sampling approach is an effective solution to the tuning problem. Ge [40] performed the comparative Analysis of SDP Algorithms in Supervised Learning Software Using Imbalanced Classification Datasets. Alan and Catal [41] proposed an Outlier Detection Method Utilizing OO Metrics Thresholds. Shivaji et. al. [42] proposed Gain Ratio based FS method suitable for classification-based defect prediction. Kumar and Rath [43] has utilized a genetic algorithm (GA) as a FS technique to identify the optimal set of source code metrics. Lu et. al. [44] observed that Multidimensional Scaling MDS which uses Random Forest similarity measure on the software metrics (independent prediction variables) outperforms other active learning approaches. Aktas and Buzluca [45] used CFS and PCA techniques to derive the most suitable subset of metrics. Yang and Qian [46] employs an automated parameter tuning method called Caret to find the best possible parameter configurations and that outperforms the default parameter settings. Malhotra et. al. [47] proposed Parameter Tuning on SDP through Differential Evolution & Simulated Annealing. Cui [48] proposed novel feature selection method based on CFS evaluator and GreedyStepwise (GS) search. Cui et. al. [49] conducted a study on the impact of the number of features on the performance of SDP models.

1.5.3 Research directions revealed from literature review

- It has been noted that in majority of the SQP models, the techniques to address the issues the performance of models are not employed.
- It has been noted that in most of the chosen primary studies that addressed the

imbalanced data, the techniques employed are very primitive. Those techniques alone are not sufficient considering the data complexities. There is scope to apply various advance techniques to build improved prediction models.

- In most of the studies, the size of public dataset used is very small. The performance of prediction models should be validated on large datasets.
- It has been reflected through this review that few to ensembles and hybridized techniques gave very good performance. The generalized findings regarding the applicability of these methods could not be presented, as these methods were hardly seen replicated. Thus future work should focus on applying these techniques on diverse datasets to establish their applicability in this area.
- Few of the selected primary studies reported the threat of result bias as the software system from which datasets have been extracted has got specific properties. Thus to reduce this threat future studies should focus on cross project validation or inter-project validation.
- Researchers have showed how the different issues have impact on the performance of quality predictors. But the studies are limited to only few machine learning algorithms. Thus, a comprehensive and comparative study has to be carried out on various machine learning algorithms.
- There are no studies found to address outliers and multi-collinearity problems in the defect prediction models built using machine learning algorithms. So it is required to analyze the possibility of occurrence of outliers and multi-collinearity problems and address them.

- Few studies have proposed Nature-inspired algorithms to address the different issues in the Machine Learning based defect predictors. So it is required to compare the performance and exploit those algorithms to develop better models.
- Some researchers have developed certain tools for collecting data sets or building models etc. They are helpful in conducting research in Software/Computer Engineering field. A study on the available tools has to be conducted. Also, new tools if desired may be developed.

1.6 Objectives of the Thesis

1.6.1 Vision

Improving the performance of software quality prediction models using Machine Learning (ML) techniques.

1.6.2 Focus

The focus of the thesis is centered on several key research objectives, each addressing a specific aspect of software quality predictive modeling. These objectives are designed to systematically explore, analyze, and improve the factors that influence the performance of predictive models, with a particular emphasis on imbalanced data, outliers, overfitting & underfitting, multi-collinearity, parameter tuning, and feature selection. The research is also focused on developing classifiers that are effective in identifying defect-proneness, change-proneness and maintainability of classes or modules, thereby enhancing the predictive

accuracy and reliability of SQP models. In this regard, effective defect prediction and defect categorization models were developed for open-source software systems. To ensure the reliability and generalizability of the predictive models, the research employs rigorous validation techniques, including ten-fold cross-validation, to minimize bias in the results. Therefore, this study specifically aims to achieve the following objectives:

1. To perform a systematic review of the classification algorithms used in software quality predictive models.
2. To conduct systematic literature review to identify and validate the different factors effecting the performance of software quality prediction models.
3. To study methods employed to categorize defects for Software Maintenance.
4. To study and apply parameter tuning techniques for software quality prediction models.
5. To study, analyze and propose methods to address data imbalance problem in software quality prediction models.
6. To review and explore feature selection techniques for software quality prediction models.
7. To study techniques used to handle multi-collinearity and apply them in software quality prediction models.
8. To explore the latest algorithms in developing software quality prediction models and evaluate their performance.

9. To study and propose metrics for assessing software quality in view of latest software development methodologies.

1.6.3 Goals

Each of the research objectives outlined above is accompanied by specific goals that detail the steps and milestones necessary to achieve the desired outcomes. These goals are designed to guide the research process, ensuring that each objective is addressed systematically and comprehensively.

1. Systematic review of classification algorithms employed for software quality prediction models

    1.1. To identify the various algorithms used in software quality predictive modeling, categorizing them based on their underlying principles and approaches.

    1.2. To analyze the performance of these algorithms in diverse contexts, identifying strengths, weaknesses, and areas for improvement.

    1.3. To highlight gaps in the existing literature and suggest potential directions for future research, focusing on the development of novel algorithms that can address existing limitations.

2. Systematic Literature Review on Factors Affecting Model Performance

    2.1. To systematically review the literature on factors affecting the performance of software quality predictive models, focusing on issues such as data imbalance, outliers, overfitting & underfitting, multi-collinearity, parameter tuning, and feature selection.

2.2. To identify and validate the most significant factors that influence model performance, providing a solid foundation for further research.

2.3. To propose a comprehensive framework that integrates these factors into a unified approach for improving predictive model performance.

3. Application of Parameter Tuning Techniques

3.1. To study existing parameter tuning techniques and their application in software quality predictive modeling.

3.2. To develop and validate new parameter tuning methods that optimize model performance, focusing on techniques that balance accuracy and generalization.

3.3. To apply these methods to real-world datasets, demonstrating their practical utility in improving predictive model performance.

4. Addressing Data Imbalance in Predictive Models

4.1. To explore existing techniques for handling imbalanced data in software quality prediction models, evaluating their effectiveness and limitations.

4.2. To develop and validate new methods for addressing data imbalance, focusing on techniques that enhance predictive accuracy without sacrificing model interpretability.

4.3. To apply these techniques to real-world datasets, demonstrating their practical applicability and effectiveness in improving model performance.

5. Exploration of Feature Selection Techniques

5.1. To review existing feature selection techniques used in software quality predictive modeling, identifying their strengths and limitations.

5.2. To develop and validate new feature selection methods that can improve model performance by choosing the most relevant and informative features.

5.3. To apply these techniques to various datasets, demonstrating their effectiveness in enhancing predictive accuracy and reducing model complexity.

6. Handling Multi-Collinearity

6.1. To review existing techniques for handling multi-collinearity in software quality predictive models, identifying their strengths and limitations.

6.2. To develop and validate new methods for mitigating the impact of multi-collinearity on predictive model performance.

6.3. To apply these techniques to various datasets, demonstrating their effectiveness in enhancing model accuracy and reliability.

7. Exploration of Latest Algorithms for Predictive Modeling

7.1. To explore and evaluate the latest AI and ML algorithms for software quality predictive modeling, focusing on their applicability and effectiveness.

7.2. To develop and validate new predictive models using these algorithms, comparing their performance with existing methods.

7.3. To perform a thorough analysis of the results, identifying the most promising algorithms for future research and application.

8. Validation of Open Source Datasets for Quality Prediction

8.1. To collect and validate datasets from open-source software projects, ensuring their suitability for quality prediction and inter-project validation.

8.2. To apply predictive models to these datasets, demonstrating their generalizability and robustness across different software projects.

8.3. To propose guidelines for dataset validation and inter-project validation, providing a roadmap for future research in this area.

9. Exploration of Software Metrics for Quality Prediction

9.1. To review existing software metrics used in quality prediction, identifying their relevance and applicability to modern software development practices.

9.2. To develop and validate new software metrics that can enhance the predictive accuracy and reliability of software quality models.

9.3. To apply these metrics to various datasets, demonstrating their effectiveness in predicting software quality in different contexts.

1.7 Overview of the Work

Software quality prediction is a critical area in software engineering, addressing the need to ensure reliability, maintainability, and efficiency in software systems. The complexity of modern software, coupled with the rapid evolution of development methodologies, has made it increasingly important to develop robust and accurate models that can predict software quality by detecting potential defects early in the development process. The overarching aim of the work was to improve the performance of prediction models in

software quality by addressing various challenges such as imbalanced data, outliers, hyperparameter tuning, feature selection, multi-collinearity, and the application of advanced machine learning and deep learning techniques. This overview presents a comprehensive summary of the research conducted, methodologies, and significant findings of each study.

A thorough systematic literature review was carried out following the guidelines provided by Kitchenham [50] to obtain a clear understanding of previous research in the area of software quality prediction. The primary objective of this review was to explore and comprehend studies on software quality prediction from the following perspectives:

- The most useful metrics in identifying the change and defect prone classes/modules.
- The machine learning algorithms effective in identifying the change and defect prone classes/modules.
- To identify the issues affecting the performance of machine learning algorithms in software quality predictive modeling.
- To study the techniques being used for handling parameter tuning.
- To study the techniques being used for handling imbalanced data.
- To study the techniques being used for handling feature selection.
- To study the techniques being used for handling multi-collinearity.
- To identify the kinds of data sets being used by researchers in prediction model development.

Once the search string was established, the most relevant and reputable digital libraries were chosen to extract pertinent papers related to the subject of software quality prediction. By thoroughly analyzing the data gathered from these studies, the research questions were addressed. Various researchers have explored the application of ML techniques in SQP. Nevertheless, further empirical studies are necessary to validate these algorithms across multiple datasets and to compare their outcomes.

Categorizing software defects based on attributes such as maintenance effort and change impact is vital for efficient resource allocation during the software maintenance phase. The research developed and validated various defect categorization models, utilizing different machine learning techniques. The first study in this domain developed Software Defect Categorization (SDC) models using the Multinomial Naïve Bayes (NBM) algorithm. The study focused on three software defect attributes: maintenance effort, change impact, and a combined approach that integrates both. Text mining techniques were employed to extract relevant features from bug reports, and the performance of the SDC models was evaluated using the Area Under the Receiver Operating Characteristic (ROC) curve. The SDC models based on the combined approach of maintenance effort and change impact exhibited superior performance compared to models based on individual attributes. The study demonstrated the efficacy of the NBM algorithm in classifying software defects. The study underscores the importance of considering multiple attributes in defect categorization models. Building on the previous study, another study was focused on developing SDC models using ensemble learning techniques. The study conducted a comprehensive evaluation of four ensemble learning techniques, namely Random Forest, XGBoost, AdaBoost, and Bagging, for software bug categorization. Experiments were conducted on five Android applications - Bluetooth, Browser, Calendar, Camera, and

MMS. Thus, a total of 5 (bug datasets) x 4 (predictor sets) x 3 (approaches) x 4 (techniques) = 240 SBC models were built during experimentation. The results of the study demonstrate that ensemble learning techniques can significantly improve the accuracy of bug categorization. Among the four techniques, Random Forest achieved the best performance, followed by Bagging, AdaBoost, and XGBoost. Thus, the study provides strong evidence that ensemble learning techniques can be effectively used for software bug categorization. Third study in this domain focused on developing SDC models using Convolutional Neural Networks (CNNs), a powerful deep learning technique. The study developed CNN-based SDC models for five Android operating system application modules. A total of 60 models were created, considering different feature sets and categorization approaches. The performance of the models was evaluated using the AUC metric. The CNN-based SDC models outperformed traditional machine learning approaches, achieving high predictive accuracy in categorizing software defects. The study also highlighted the robustness of CNNs in handling large and complex datasets. Overall, this work illustrates the potential of deep learning techniques to enhance the accuracy and efficiency of SDC. However, additional research is required to enhance the performance of deep learning architectures on high-level faults and to explore other deep learning architectures and techniques for SDC.

One of the significant contributions of the work is the systematic review of hyperparameter tuning techniques for software quality prediction models. Hyperparameter tuning is critical for optimizing the performance of ML models, yet it has often been overlooked in the context of software quality prediction. Hyperparameters are critical configurations for ML algorithms that significantly influence the performance of predictive models. In software quality prediction, models are employed to detect vulnerable

software components early in the development process, aiding in better resource allocation and enhancing overall software quality. However, many studies rely on default hyperparameter settings, which can lead to suboptimal model performance. The study conducted a systematic review of existing literature to identify and analyze studies that have utilized hyperparameter tuning techniques in software quality prediction. The review covered various domains, including defect prediction, maintenance estimation, change impact prediction, reliability prediction, and effort estimation. The review identified 31 primary studies on hyperparameter tuning for software quality prediction models. The findings highlighted that tuning hyperparameters significantly enhances the predictive accuracy of models. Additionally, it was observed that certain classification algorithms are highly sensitive to their parameter settings, achieving optimal performance when tuned appropriately. Conversely, some algorithms exhibit low sensitivity to hyperparameters, making tuning unnecessary in such cases. The study concluded that hyperparameter tuning is essential for improving the predictive capability of software quality models. Practical guidelines were provided to facilitate effective hyperparameter tuning, offering insights for both researchers and practitioners in the field.

Imbalanced data is a pervasive challenge in machine learning, particularly in software defect prediction, where the number of defective software components is often significantly lower than non-defective ones. This research explored various techniques to address this issue, focusing on Artificial Neural Networks (ANNs) to improve model performance. The study investigated the effectiveness of various data resampling techniques in improving the performance of ANN-based SDP models. A total of twelve data sampling methods, including over-sampling, under-sampling, and hybrid techniques, were applied to six distinct defect datasets from open-source Java-based systems. The performance of the

resulting 78 SDP models was assessed using ten-fold cross-validation and measures such as AUC, G-Mean, and Balance. The study found that handling imbalanced data significantly improves the performance of ANN-based SDP models. The Synthetic Minority Oversampling with Edited Nearest Neighbor Technique (SMOTE-ENN) outperformed other techniques, demonstrating its effectiveness in addressing imbalanced data. The research advocates for the use of oversampling and hybrid data balancing techniques in developing effective defect prediction models. The findings provide a foundation for further exploration of imbalanced learning techniques in SDP. To further address the issue of imbalanced data, another study in this research proposed the use of a Weighted Loss Function for Neural Networks (WL-NN). Four types of defect prediction models were constructed: NN over imbalanced data, WL-NN over imbalanced data, NN over balanced data, and WL-NN over balanced data. The experiments were carried out on 22 open-source datasets from the AEEEM, JIRA, and PROMISE repositories. The results demonstrated that the proposed WL-NN significantly improves the performance of SDP models. When combined with data resampling techniques, WL-NN outperformed other approaches, achieving the highest predictive performance. The study strongly recommends the adoption of the WL-NN approach for handling imbalanced data in software defect prediction. The findings contribute to the ongoing research on improving the robustness and accuracy of predictive models in the presence of imbalanced data.

Feature selection is a crucial step in building effective predictive models, as it helps in identifying the most relevant features while reducing the dimensionality of the dataset. This research explored the application of swarm intelligence techniques for feature selection in SDP. This study focused on evaluating the effectiveness of swarm intelligence techniques, particularly Cuckoo Search (CS) and Crow Search (CRS), in selecting relevant features

for software defect prediction. The research employed a diverse set of feature selection methods and machine learning algorithms on multiple datasets from the PROMISE repository. Comparative analysis was conducted using traditional filter-based techniques such as chi-square and information gain, alongside the swarm intelligence techniques. The findings revealed that CS and CRS outperformed traditional filter-based methods in selecting relevant features for defect prediction. These swarm intelligence techniques demonstrated superior performance in exploring the solution space and identifying high-quality feature subsets. The study emphasizes the potential of swarm intelligence techniques to improve accuracy and efficiency of defect prediction models. The complementary nature of these techniques with traditional classifiers such as logistic regression, support vector machine, Naïve Bayes, and random forest was also emphasized.

Multi-collinearity issignificant challenges in software quality prediction, affecting the reliability and interpretability of the models. This research conducted a systematic review of the challenges and solutions related to these issues. The review identified several techniques for addressing multi-collinearity, including dimensionality reduction methods such as Principal Component Analysis (PCA), regularization techniques like Ridge and Lasso regression. The study also emphasized the importance of careful feature selection in minimizing multi-collinearity. The research found that handling these issues significantly enchance the performance of SQP models.

This research also proposed a Metric Suite for Event-Driven Software Systems. Event-driven software systems have gained significant prominence due to their ability to handle complex and asynchronous interactions. Evaluating the quality and characteristics of such systems is crucial to ensure their reliability and efficiency. This study proposed a comprehensive set of metrics specifically designed to measure event-driven software

systems. Beginning with an exposition of the event-driven programming paradigm, this study emphasizes its pivotal traits, highlighting the distinctiveness of event-driven systems in contrast to structured and object-oriented programming paradigms. The purpose of this study is to address the necessity for metrics tailored to event-driven systems. The proposed metrics are grouped into categories, such as event structure, event dependency, event performance, event complexity, event synchronization and event reliability metrics. Each metric is defined and described. These metrics facilitate software practitioners to make informed decisions during system design, optimization, and evaluation processes. This work concludes by discussing the limitations of the study, including potential threats to its validity. Future guidelines are also outlined, highlighting opportunities for further research, industry adoption, tooling, benchmarking, and continuous improvement of the proposed metrics.

1.8 Organization of the Thesis

This section outlines the structure of the thesis. The thesis is structured into ten chapters, each focusing on a specific aspect of the research undertaken to develop and validate improved machine learning techniques for software quality predictive modeling. The organization of the thesis is as follows: **Chapter 1** sets the stage for the entire research work by presenting the basic concepts of work and the motivation of the thesis. **Chapter 2** details the research methodology adopted to achieve the research objectives. **Chapter 3** presents a comprehensive systematic literature review conducted according to established guidelines to identify the research gaps. **Chapter 4** presents the construction of software defect categorization models based on maintenance effort and change impact. **Chapter 5**

presents the hyperparameter tuning techniques in software quality prediction. **Chapter 6** proposes the techniques to handle imbalanced data in software quality prediction. **Chapter 7** proposes swarm intelligence-based approaches for feature selection in software quality prediction. **Chapter 8** analyzes the techniques to handle the problems of multi-collinearity, overfitting & underfitting and outliers in software quality prediction. **Chapter 9** proposes a metric suite for event-driven software systems. Finally, **Chapter 10** presents the conclusions of the thesis. The brief description of each chapter is given below.

Chapter 1: The introduction sets the stage for the entire research work. It begins by discussing the significance of software quality prediction in the context of software engineering. The chapter outlines the challenges associated with predictive modeling in software quality, including imbalanced data, outliers, multi-collinearity, and the need for efficient feature selection and parameter tuning. The chapter also presents the research objectives, the scope of the study, and the contributions of the research. This chapter also describes the detailed steps of developing quality prediction models.

Chapter 2: This chapter details the research methodology adopted to achieve the research objectives. It begins with a discussion on the design of the study, including the selection of datasets, the choice of algorithms, and the evaluation metrics used to assess model performance. The chapter also covers the experimental setup, data preprocessing techniques, and the steps involved in the development and validation of predictive models. Finally, the chapter discusses the systematic approach taken to address the challenges identified in the research.

Chapter 3: The third chapter presents a comprehensive systematic literature review conducted according to established guidelines. This review provides a detailed understanding of the existing research in software quality prediction, with a focus on areas such as

defect prediction, change impact analysis, and maintainability estimation. The review highlights the strengths and limitations of current approaches, identifies research gaps, and sets the foundation for the subsequent chapters. Key findings from the review are used to justify the need for improved techniques in software quality prediction.

Chapter 4: This chapter focuses on the categorization of software defects based on maintenance effort and change impact. It presents four distinct studies that employ different machine learning techniques for defect categorization: Multinomial Naïve Bayes (NBM), ensemble methods, Convolutional Neural Networks (CNN), and deep learning methods. The chapter compares the effectiveness of these approaches and highlights the combined impact of maintenance effort and change impact on defect categorization.

Chapter 5: This chapter delves into the importance of hyperparameter tuning in enhancing the performance of predictive models. It provides an in-depth analysis of different tuning techniques, including grid search, random search, and more advanced methods. The chapter presents the results of experiments that show how tuning hyperparameters can significantly improve model performance. Practical guidelines for hyperparameter tuning in software quality prediction are also provided, based on the insights gained from the experiments.

Chapter 6: This chapter addresses the challenge of imbalanced data in software defect prediction. It explores various techniques to manage this issue, including oversampling, undersampling, and hybrid methods. The chapter specifically focuses on the application of these techniques to Artificial Neural Networks (ANN) and evaluates their effectiveness in improving model performance. This chapter also introduces a novel Weighted Loss Function for Neural Networks (WL-NN) designed to tackle imbalanced data in software defect prediction. The chapter presents experimental results demonstrating the superiority

of the WL-NN approach in enhancing prediction accuracy.

Chapter 7: This chapter investigates the application of swarm intelligence techniques for feature selection in software defect prediction. It compares these techniques with traditional filter-based methods and assesses their effectiveness in improving the accuracy of defect prediction models. The chapter demonstrates how swarm intelligence can be leveraged to identify optimal feature subsets, enhancing model performance.

Chapter 8: This chapter delves into the issue of multi-collinearity in software quality prediction models. It reviews the causes and consequences of multi-collinearity and discusses various techniques to mitigate its impact. The chapter provides practical guidelines for managing these issues, ensuring the development of reliable and accurate prediction models.

Chapter 9: This chapter proposes a comprehensive set of metrics tailored for evaluating event-driven software systems. It discusses the unique characteristics of event-driven systems and the necessity for specialized metrics to assess their quality. The chapter introduces various metric categories, including event structure, event dependency, and event performance, providing a detailed explanation of each metric and its application.

Chapter 10: The final chapter summarizes the key findings and contributions of the thesis. It discusses the implications of the research for both academia and industry, highlighting the advancements made in software quality prediction. The chapter also identifies potential areas for future research, offering suggestions for further exploration and development in this field.

Chapter 2

Research Methodology

2.1 Introduction

In modern software engineering, predictive models serve a crucial role in assessing the quality of software systems. These models leverage historical data to forecast future software defects, anticipate changes, and evaluate maintainability. The purpose of the research is to improve the performance of machine learning algorithms in identifying defect-prone or change-prone software classes/modules.

This chapter on research methodology provides a comprehensive framework for the systematic investigation carried out during the research. It outlines the strategies, methods, and techniques employed to achieve the objectives of the study. This chapter includes the definition of the research problem, the selection of variables, data collection procedures, and the experimental design framework. Through this, the methodology ensures the validity, reliability, and rigor of the research findings, guiding how the software quality

predictive models were developed, validated, and analyzed.

The structure of this chapter is as follows: it begins by outlining the research process in Section 2.2, followed by the problem definition in Section 2.3, the methodology for literature review in Section 2.4, a detailed explanation of the variables in Section 2.5, the data analysis methods employed in Section 2.6, the experimental design framework in Section 2.7, the empirical data collection in Section 2.8, the model development and validation in Section 2.9, the performance measures employed in Section 2.10 and finally, statistical analysis in Section 2.11.

2.2 Research Process

The research process followed in this study involves several stages, each designed to methodically address the research objectives and hypotheses [51]. The stages of the research process are summarized in Figure 2.1 below:

1. **Problem Definition:** Identifying the research problem, including challenges in current software quality prediction models, such as imbalanced data and multi-collinearity.

2. **Literature Review:** Conducting a systematic review of past research to identify existing gaps, algorithms, and best practices in software quality prediction.

3. **Data Collection:** Collecting datasets from open-source repositories (PROMISE[52], AEEEM [53], JIRA[54] , and Android Modules[55]) for software quality prediction.

4. **Preprocessing and Data Balancing:** Applying data cleaning, normalization, and

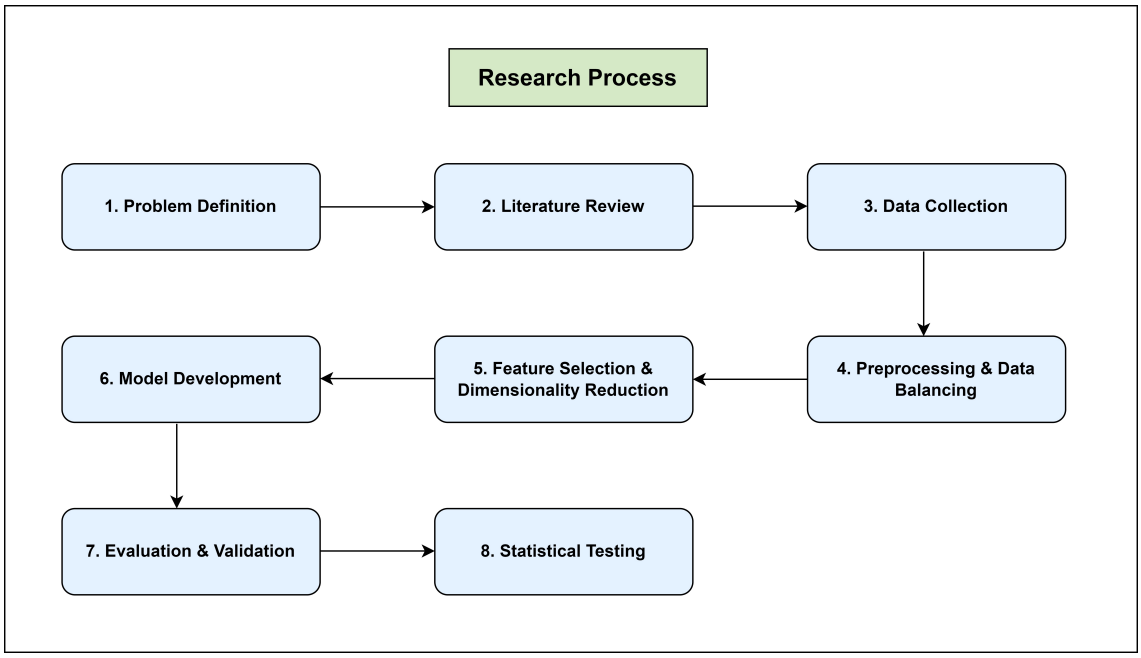

Figure 2.1: Research Process

balancing techniques such as SMOTE (Synthetic Minority Over-sampling Technique) to handle imbalanced datasets.

5. **Feature Selection and Dimensionality Reduction:** Using methods such as Recursive Feature Elimination (RFE) and Principal Component Analysis (PCA) to select relevant features and reduce dimensions.

6. **Model Development:** Developing predictive models using machine learning algorithms, including Naïve Bayes, Ensemble Methods, Convolutional Neural Networks (CNNs), etc.

7. **Evaluation and Validation:** Evaluating the performance of models using met-

rics like AUC (Area Under the Curve), G-Mean, Balance, and MCC (Matthews Correlation Coefficient).

8. **Statistical Testing:** Employing statistical tests such as the Friedman test and Wilcoxon Signed-Rank test to verify the significance of performance improvements.

2.3 Define Research Problem

The primary research problem is to improve the performance of software quality prediction models by addressing key challenges, including hyperparameter tuning, imbalanced data, feature selection, and multi-collsinearity. This research seeks to answer the following research questions:

- How can machine learning algorithms be improved to better identify defect-prone or change-prone modules?
- What are the most effective approaches for categorization of software defects?
- What data preprocessing techniques can mitigate the impact of imbalanced datasets on predictive models?
- Which feature selection methods yield the best results in terms of predictive accuracy?
- How can overfitting and underfitting be controlled through effective parameter tuning and model optimization?

- How can multi-collinearity be addressed in software quality prediction models to improve stability and accuracy?
- What new software metrics are needed to assess software quality in the context of modern programming paradigms?

2.4 Literature Survey

This section delves into the current state of research in software quality prediction. The literature survey helps in understanding the progress made in the field, identifying gaps in current research, and discovering opportunities for further advancements [56]. This provides the foundation for addressing the research problem by drawing insights from past studies, methodologies, and experiments that have attempted to solve similar problems. This is essential in identifying the tools, techniques, algorithms, and challenges faced in software quality predictive modeling.

2.4.1 Steps to Conduct a Literature Survey

i. Define the Scope of the Survey

- Start by clearly identifying the key research questions and themes relevant to the research, such as the improvement of predictive models for software defects, addressing data imbalance, or multi-collinearity.
- Delimit the scope based on years, relevance to machine learning, software defect prediction, and the specific challenges that have been identified.

ii. Source Identification

- Identify credible sources, including peer-reviewed journal articles, conference papers, technical reports, and open-source datasets like PROMISE and AEEEM. Databases like IEEE Xplore, Springer, ACM Digital Library, and Google Scholar are crucial.

iii. Keyword Search

- Develop a comprehensive list of keywords related to the research (e.g., ”software defect prediction”, ”multi-collinearity in machine learning”, ”SMOTE in software quality models”, ”convolutional neural networks in defect prediction”).

iv. Organize and Filter Results

- Filter articles based on relevance, citations, and recency. Focus on studies that provide insights into the algorithms, metrics, and methodologies related to software quality and defect categorization.
- Use tools like EndNote or Mendeley for citation management and organization.

v. Critical Evaluation of Sources

- Critically assess each source by examining their methodologies, data collection procedures, and results. Pay attention to studies that have successfully tackled the challenges relevant to the research.
- Assess the performance metrics used (AUC, G-Mean, Balance) and the statistical tests employed in their validation.

vi. Synthesizing Information

- Combine findings from different studies to highlight common challenges, successful methodologies, and areas where further research is required.
- Organize the findings into categories such as ”Data Imbalance Solutions”, ”Handling Multi-collinearity”, ”Feature Selection Techniques”, and ”Performance Metrics in Software Quality Prediction”.

vii. Identify Research Gaps and Opportunities

- Based on the review, pinpoint the areas where improvements are needed, such as more efficient algorithms for handling outliers, techniques for addressing overfitting/underfitting, or the development of new metrics for modern programming paradigms.

viii. Writing the Literature Review

- Structure the literature review into thematic sections that align with the objectives of the research. Each section should focus on addressing a key challenge, highlighting past efforts, and providing a rationale for the proposed approach.

2.4.2 Systematic Review of Classification Algorithms in SQP

Several classification algorithms have been employed in software quality prediction, including decision trees, support vector machines (SVM), neural networks, and ensemble methods like Random Forests. This literature survey is conducted to highlight several

issues in existing predictive models, such as limited handling of imbalanced data and difficulty in optimizing algorithms for better performance across varied software systems.

2.4.3 Systematic Review of Data Imbalance Problem in SQP

Data imbalance refers to a disproportionate representation of classes, where one class (e.g., defect-prone modules) is much smaller than the other (non-defect-prone modules). Imbalanced data can severely impact the performance of ML algorithms by biasing them toward the majority class. This literature survey is conducted to study the different approaches to address these challenges.

2.4.4 Systematic Review of Parameter Tuning Techniques in SQP

Parameter tuning methods optimize hyper-parameters of algorithms, thereby preventing overfitting and underfitting. This literature survey is conducted to study the different techniques employed to optimize hyper-parameters.

2.4.5 Systematic Review of Feature Selection Techniques and Multi-collinearity in SQP

Feature selection is critical to reduce problems of multi-collinearity and improve model interpretability. This literature survey is conducted to study the different techniques employed to address these challenges.

2.5 Defining Variables

Variables are defined as elements or factors that can change or be changed in an experiment or study[57]. The variables in prediction models are divided into independent and dependent variables. Understanding these variables is crucial for building accurate models that can generalize across various software systems.

2.5.1 Independent Variables

Independent variables are those that are manipulated or controlled in the research to observe their effect on other variables [58]. They are considered the ”cause” in a cause-and-effect relationship.

The Object-Oriented Metrics serve as independent variables in this research. These metrics represent characteristics of software modules and are strong indicators of potential defects or changes in the codebase [59]. Below are some of the commonly used object-oriented metrics:

- *Lines of Code (LOC):* The total number of lines in a software module.
- *Cyclomatic Complexity (CYC):* Measures the number of linearly independent paths through a program's source code, indicating the complexity of the control flow.

$$CC = E - N + 2P \tag{2.1}$$

where E is the number of edges in the control flow graph, N is the number of nodes and P is the number of connected components.

- *Depth of Inheritance Tree (DIT):* Measures how deep a class is in the inheritance hierarchy. A deeper inheritance tree can complicate understanding and maintenance, potentially increasing defect rates.
- *Number of Children (NOC):* Quantifies the number of immediate subclasses derived from a superclass. A higher NOC suggests a more complex class hierarchy, influencing maintainability.
- *Number of Public Methods (NPM):* Counts the number of public methods in a class. A higher count may suggest more interactions with other classes, increasing complexity.
- *Methods per Class (MPC):* Measures the average number of methods defined in each class. A high MPC can indicate increased complexity affecting maintainability.

$$\text{MPC} = \frac{\text{Total Number of Methods}}{\text{Total Number of Classes}} \tag{2.2}$$

- *Weighted Methods per Class (WMC):* Sums the complexities of all methods in a class, serving as a measure of overall class complexity. Higher WMC values may correlate with higher chances of defects.

$$\text{WMC} = \sum_{j=1}^{m} \text{Complexity}(M_j) \tag{2.3}$$

  where M_j represents each method and m is the total number of methods in the class.

- *Response for a Class (RFC):* Counts the number of methods that can be executed in response to a message sent to an object of that class. A higher RFC indicates a more

complex class that may be harder to test and maintain.

$$\text{RFC} = |\text{M}| + |\text{C}| \tag{2.4}$$

where $|\text{M}|$ is the number of methods in the class and $|\text{C}|$ is the number of methods in the classes called by the methods of this class.

- *Lack of Cohesion of Methods (LCOM):* Assesses the cohesion within a class by measuring how well the methods of the class are related. Low cohesion may indicate a class trying to perform too many unrelated tasks, potentially leading to higher defect rates.

$$\text{LCOM} = \begin{cases} 0 & \text{if all methods access the same instance variables} \\ \text{Count of Disconnected Methods} & \text{otherwise} \end{cases} \tag{2.5}$$

- *Coupling Between Objects (CBO):* Measures the interdependencies between classes and is a strong indicator of code complexity and maintenance effort.

$$\text{CBO} = \sum_{i=1}^{n} |\text{M}_i| \tag{2.6}$$

where M_i represents each class that is referenced by the class being measured, n is the total number of classes that are directly coupled to the class.

- *Class Interface (CI):* Evaluates the number of public interfaces provided by a class. A high number may increase the likelihod of misuse and defects.

- *Fan-in and Fan-out:* Fan-in measures the number of classes that call methods of a particular class, while fan-out counts the number of classes that a given class calls. High fan-in can indicate a class is well-utilized, while high fan-out may suggest excessive dependencies.

$$\text{Fan-in} = \text{Number of Classes Calling this Class} \tag{2.7}$$

$$\text{Fan-out} = \text{Number of Classes Called by this Class} \tag{2.8}$$

These independent variables are known to affect software quality and are thus used as predictors in the machine learning models developed.

2.5.2 Dependent Variable

Dependent variables are those that are measured or observed in response to changes in the independent variables. They are considered the ”effect” in a cause-and-effect relationship.

The dependent variable in this research is a binary or categorical label indicating whether a software module is defect-prone or change-prone. In this research, the dependent variable is formulated as follows:

- *Defect Proneness:* The module is labeled as defect-prone (1) if defects were identified in past versions, and non-defect-prone (0) otherwise.

- *Change Proneness:* Similarly, change-proneness refers to the likelihood that a module will require modification in future software releases.

2.6 Data Analysis Methods

Data analysis in this research follows a rigorous pipeline that includes data preprocessing, feature selection, and applying machine learning algorithms. Several data analysis techniques were employed to ensure the accuracy and robustness of the predictive models.

2.6.1 Data Preprocessing

Data preprocessing is the first critical step in data analysis. The following steps were undertaken during the preprocessing phase:

- **Handling Missing Data:** Imputation techniques such as mean or mode imputation were applied to fill in missing values.
- **Normalization:** Features were normalized using Min-Max Scaling to ensure all features contribute equally to the model:

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{2.9}$$

- **Outlier Detection:** Outliers were identified using Z-scores and handled appropriately to prevent model distortion.
- **Data Transformation:** Transformation techniques like log transformation were used to handle skewed distributions of independent variables.

2.6.2 Data Balancing

Given that software defect data is often imbalanced, with significantly more non-defect-prone modules than defect-prone ones, data balancing techniques are essential.

2.6.3 Feature Selection

Feature selection methods, including Recursive Feature Elimination (RFE) and Principal Component Analysis (PCA), were used to reduce the dimensionality of the dataset while retaining the most informative features. Feature selection helps mitigate the issue of multi-collinearity and enhances the model's generalizability.

2.6.4 Classifiers

The study employs the following machine learning techniques for quality prediction:

2.6.4.1 Logistic Regression (LR)

Logistic regression is a linear classification algorithm that models the probability of a binary outcome based on one or more predictor variables. The logistic regression formula is given by:

$$P\left(y=\frac{1}{X}\right)=\ \frac{1}{1+\ e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2+....+\beta_n x_n)}} \tag{2.10}$$

$P\left(y=\frac{1}{X}\right)$ is the probability of the dependent variable *y* being 1 given the predictor variables X_1,X_2,... X_n and $\beta_0\ ,\ \beta_1, \ldots., \beta_n$ are the coefficients of the logistic regression model.

2.6.4.2 Support Vector Machine (SVM)

Support vector machine is a powerful classification algorithm that constructs hyperplanes in a high-dimensional space to separate data points of different classes. The SVM formula for binary classification is given by:

$$f(x) = sign(\sum_{i=1}^{N} \alpha_i y_i \, K\,(x, x_i) + b) \tag{2.11}$$

where *f(x)* is the decision function, α_i are the Lagrange multipliers, y_i are the class labels, $K\,(x, x_i)$ is the kernel function, and *b* is the bias term.

2.6.4.3 Naïve Bayes (NB)

Naïve Bayes is a probabilistic classifier based on Bayes' theorem with the assumption of independence between features. The Naïve Bayes formula for classification is given by:

$$P(y/X) \; = \frac{P(X/y)P(y)}{P(X)} \tag{2.12}$$

where $P(y/X)$ is the posterior probability of class *y* given the features *X*, $P(X/y)$ is the likelihood of the features given the class, *P(y)* is the prior probability of class *y*, and *P(X)* is the probability of the features.

Naïve Bayes is a simple yet effective probabilistic classifier widely used in machine learning, particularly for text classification [60]. It is based on Bayes' theorem and the ”naive” assumption of conditional independence among features. In classification, it's used to calculate the probability of a given data point belonging to a particular class.

Let’s define:

- $p(data)$*:* Probability of the data point (constant for all classes).

- $p(class)$*:* Prior probability of the class. It's calculated as the ratio of the number of data points in that class to the total number of data points.

$$p(class) = \frac{Number\ of\ datapoints\ in\ class}{Total\ number\ of\ data\ points} \tag{2.13}$$

- $p\left(\frac{feature_i}{class}\right)$: Probability of feature *i* occurring given the class.

- $p(data/class)$*:* Probability of the data point given that it belongs to the class. This represents the likelihood of observing a specific set of features given a class. In text classification, for instance, it's often calculated based on the frequency of words in documents of that class. It can be calculated as:

$$p(data/class) = p\left(\frac{feature_1}{class}\right) * p\left(\frac{feature_2}{class}\right) * \ldots\ldots * p(\frac{feature_n}{class}) \tag{2.14}$$

- $p(class/data)$*:* Probability of the data point belonging to a specific class given its features. It's calculated using Bayes' theorem.

$$p(class/data) = \frac{p(data/class) * p(class)}{p_{(data)}} \tag{2.15}$$

Naïve Bayes is computationally efficient and works well when the assumption of feature independence is reasonable. While the ”naive” assumption may not hold in all cases, Naïve

Bayes can still provide surprisingly good results, especially for high-dimensional data like text documents.

The sci-kit machine learning framework [61] in python is used to implement the Naïve Bayes classifier (sklearn.naive_bayes. GaussianNB), with default parameter configurations.

2.6.4.4 Random Forest (RF)

Random forest is an ensemble learning algorithm that constructs multiple decision trees and combines their predictions through voting or averaging. The Random Forest algorithm is based on the principle of bagging and uses random sampling of features to build each tree. The RF's final output is determined by computing the mode of the outputs generated by the constituent decision trees within the forest.

- *Bootstrap Sampling:* Random Forest starts by creating multiple bootstrap samples from the original training data. This means that for each tree in the forest, a new dataset is generated by randomly selecting samples from the original dataset with replacement. This results in multiple subsets of the data with potentially some duplicates and some samples not included. Given a dataset of size N, we randomly sample N data points with replacement to create a new dataset. This can be represented as:

$$NewDataSet = SampleWithReplacement(OriginalDataSet, N) \tag{2.16}$$

- *Decision Trees:* For each bootstrap sample, a decision tree is trained. These decision trees are often referred to as ”weak learners” because they might not perform well

individually on the entire dataset, but they capture different patterns within their respective bootstrap samples.

- *Voting (Classification) or Averaging (Regression):* When making predictions, each decision tree in the forest casts a ”vote” (in classification problems) or produces a prediction (in regression problems). For classification tasks, the class that receives the most votes becomes the final prediction. For regression tasks, the predictions from each tree are averaged. In a classification problem, each tree provides a prediction for the class label. The final prediction is determined by majority voting among all trees. Let's denote the predicted class by Tree_i as ”Class_i” for the i-th tree. The final prediction is:

$$FinalPrediction = Mode(\{Class\_1, Class\_2, ..., Class\_n\} \tag{2.17}$$

The sci-kit machine learning framework [61] in python is used to implement the random forest classifier (sklearn.ensemble.RandomForestClassifier).

2.6.4.5 XGBoost

eXtreme Gradient Boosting or XGBoost (XGB), is an ensemble method that employs a gradient boosting framework. It aims to optimize a differentiable loss function by iteratively adding decision trees [62]. The prediction of the model is the weighted sum of the predictions from the individual trees.

Let's define the set of base models as $\{T_1, T_2, ..., T_n\}$, where T_i represents individual decision trees. The final prediction, $\widehat{y}_{XGB}$, is determined by weighted averaging:

$$\widehat{y}_{XGB} = \sum_{i=1}^{n} \alpha_i \, T_i \tag{2.18}$$

where α_i represents the weight assigned to tree T_i.

2.6.4.6 AdaBoost

AdaBoost, short for Adaptive Boosting, is an ensemble method that assigns different weights to training instances based on their classification accuracy [63]. It sequentially builds a series of base models and adjusts the weights of misclassified instances. The final prediction is the weighted sum of base models.

Let's define the set of base models as $\{H_1, H_2, ..., H_n\}$, where H_i represents individual models. The final prediction, $\widehat{y}_{ADB}$, is determined by weighted averaging:

$$\widehat{y}_{ADB} = \sum_{i=1}^{n} \beta_i \, H_i \tag{2.19}$$

where β_i represents the weight assigned to tree H_i.

2.6.4.7 Bagging

Bootstrap Aggregating, or Bagging (BAG), is an ensemble technique that creates multiple subsets of the training data through bootstrapping and trains independent base models on each subset [64]. The final prediction is obtained through averaging or majority voting.

Let's define the set of base models as $\{M_1, M_2, ..., M_n\}$, where M_i represents individual models. The final prediction, $\widehat{y}_{BAG}$, is determined by averaging:

$$\widehat{y}_{BAG} = \frac{1}{n}\sum_{i=1}^{n} M_i \tag{2.20}$$

Bagging with decision trees is a specific type of bagging that uses decision trees as base learners. Bagging with decision trees is a popular ensemble learning technique for classification tasks [64].

2.6.4.8 Artificial Neural Network(ANN)

Artificial Neural Networks(ANNs), or simply, Neural Networks (NNs) represent a powerful machine learning methodology inspired by the intricate mechanisms of the human brain. Robert Hecht-Nielsen defined "neural network as a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs" [65]. Similar to the human brain, ANN comprises interconnected neurons or nodes that are arranged into various layers within the network. The neurons in an ANN function by processing information and transmitting it to other neurons in the network. The structure of ANNs is designed to enable them to learn and generate predictions by identifying patterns in input data. Within a Neural Network, each neuron obtains inputs from the preceding layer, which are then individually multiplied by their associated weights and subsequently summed. This weighted sum undergoes an activation function, leading to the generation of an output signal. This output signal is then transmitted to the subsequent layer, facilitating the flow of information and computation throughout the network. The output of a neuron in the i^{th} layer is given by the following equation:

$$\widehat{y_i} \;=\; _i \left(\sum_{j\,=\,1}^{n} [x_j \;\times\; w_{ij}] \;+\; b_i \right) \tag{2.21}$$

Here x_j is the j^{th} input, w_{ij} is the weight between the j^{th} input and the current neuron, n is the total number of inputs to the neuron from the preceding layer, b_i is the bias term of the neuron and $_i$ is the activation function for the current neuron. For each j in the range $[1,\ n]$, the multiplication of x_j with w_{ij} is summed up. The bias term is then added to this summation. Finally, the activation function $_i$ is employed to transform this result into the output $\widehat{y_i}$. The activation function within a Neural Network plays a pivotal role in capturing and effectively modeling complex non-linear relationships. When it comes to predicting software defects, the relationship between software metrics and component defect proneness is frequently intricate and characterized by non-linearity. Thus, the utilization of a Neural Network emerges as a suitable approach for accurate software defect prediction [66].

2.6.4.9 Convolutional Neural Network(CNN)

CNNs represent a category of deep learning models extensively utilized for the analysis of image and sequential data, including text classification. The core of a CNN consists of convolutional layers designed to learn and apply filters to the input data, scanning through it to detect patterns or features [67]. In the case of text data, these filters are typically one-dimensional. The convolution process in a CNN entails element-wise multiplication of a small filter (also known as a kernel) with the input data, resulting in the creation of a feature map. This mathematical operation can be expressed as follows:

$$S(j) = (X * W)(j) = \Sigma_{i=0}^{F-1} X(j+i) * W(i) \tag{2.22}$$

where S(j): Value of the feature map at position j, X: Input data, W: Filter (kernel) weights, and F: Filter size.

Following the convolutional layers, pooling layers are frequently employed to reduce data dimensionality while preserving crucial features. Max-pooling, a common technique, involves selecting the highest value within a small region. The mathematical representation for one-dimensional max-pooling is as follows:

$$P(j) = \max_{i=0}^{K-1} C(j+i) \tag{2.23}$$

where P(j): Output value after max-pooling, C: Feature map, and K: Pooling window size.

Subsequent to the convolutional and pooling layers, CNNs generally incorporate one or more fully connected layers, akin to traditional neural network layers found in Multi-Layer Perceptrons (MLPs). For developing Software Defect Categorization (SDC) models, the study employed back-propagation learning, a technique that optimizes the network's weights to minimize the disparity between observed and desired outputs.

$$z_{ij}^{l} = \Sigma_{p=0}^{m-1} \Sigma_{q=0}^{n-1} x_{i+p,j+q}^{l-1} w_{p,q}^{l} + \ b_{j}^{l} \tag{2.24}$$

$$\widehat{y}_{ij}^{l} = \ ^{l}\left(z_{ij}^{l}\right) \tag{2.25}$$

where $x_{i,j}^{l-1}$ represents the output of the preceding layer at position (i,j), $w_{p,q}^{l}$ represents

the filter at position (p,q) in layer l, b_j^lrepresents the bias term for neuron j in layer l, $^l(.)$ represents the activation function in layer l and $\widehat{y}_{ij}^l$ represents the output of neuron at position (i,j) in layer l.

The rationale behind choosing these machine learning techniques is their effectiveness, versatility, and widespread use in classification tasks, including defect prediction.

2.7 Experimental Design Framework

The experimental design framework is a critical part of any scientific inquiry. It serves as the blueprint for the collection, analysis, and interpretation of data, providing a systematic approach to understanding the research problem. For this research, the goal was to develop and validate software quality prediction models that address challenges like imbalanced data, outliers, overfitting, underfitting, multi-collinearity, and parameter tuning, among others. The various steps of the framework are illustrated in Figure 2.2.

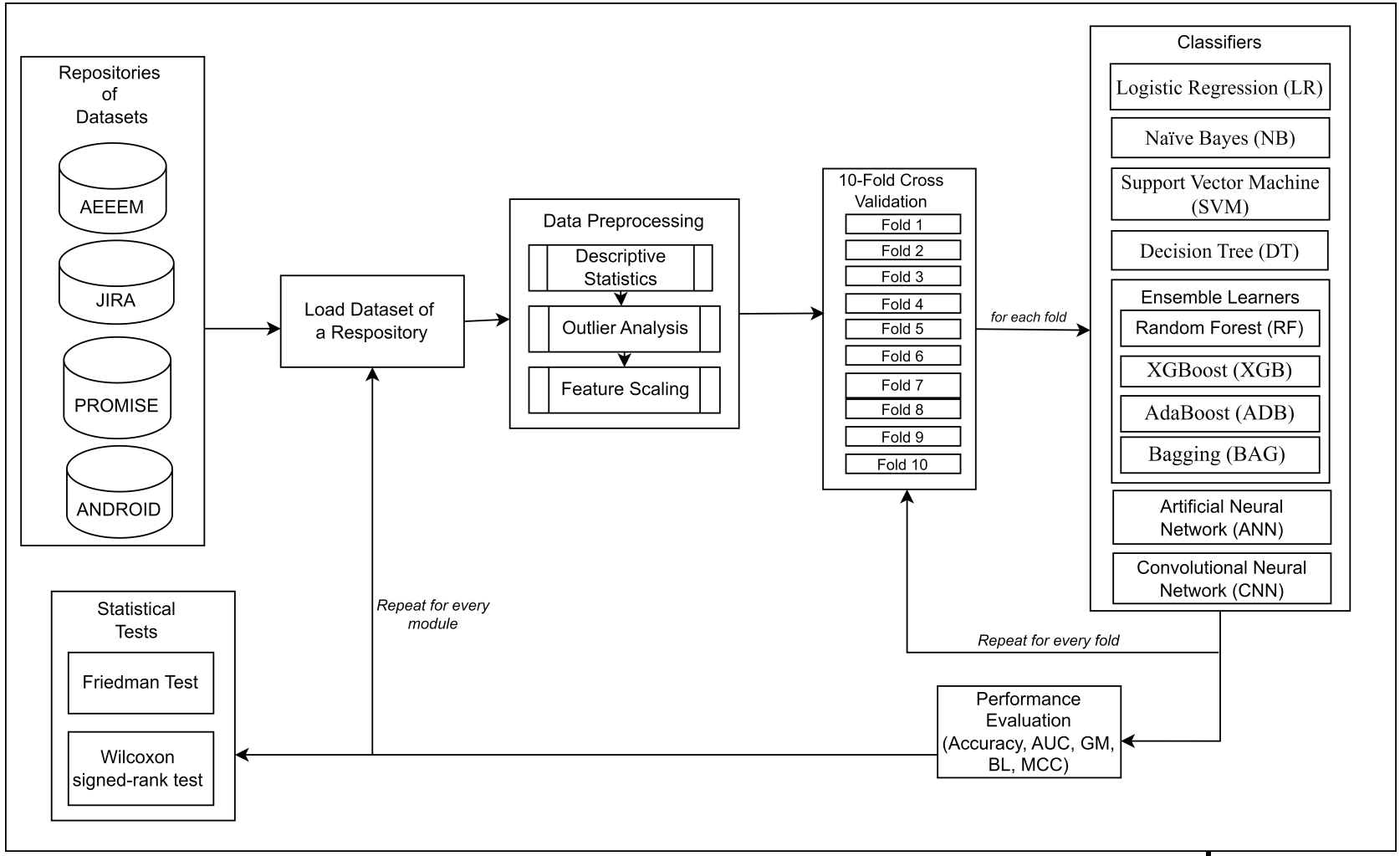

Figure 2.2: Illustration of Proposed Approach

2.8 Empirical Data Collection

Data collection is the foundation of predictive model development. For software quality prediction models, empirical data on software metrics and defect labels are required. This research utilizes publicly available datasets like PROMISE, AEEEM, JIRA repositories, and Android modules. Each of these repositories offers a variety of software projects with extensive historical data, including defect records, code changes, and object-oriented metrics.

Empirical data collection was conducted through a structured process:

i. Selection of Datasets: The PROMISE, AEEEM, JIRA and Android repositories were selected for their widespread use in the software quality research community. These datasets contain a mixture of open-source projects, which provide real-world data necessary for developing robust models.

ii. Data Extraction: Data were extracted from the selected datasets. For instance, in the PROMISE dataset, data were preprocessed by using specific software metrics as independent variables and labeling defect-prone classes as the dependent variable. Each project dataset was treated as a separate entity to maintain the consistency of empirical research.

iii. Data Storage: Extracted data were stored in a structured format (e.g., CSV, ARFF, or SQL databases) to ensure easy access during the preprocessing and model development phases. Each dataset was labeled clearly with the corresponding software project and version information.

2.8.1 Datasets

Quality prediction models are developed using datasets from defect repositories such as AEEEM[53], JIRA[54], PROMISE[52], and Android[55]. The dataset consists of independent variables, such as software metrics and process metrics. These variables are required for the training of a model and to complete the predictive model task. These software metrics consist of McCabe's cyclomatic metrics, C&K metrics, and other OOM. All of the datasets have a different number of independent variables. These software metrics are characterized based on different measures such as cohesion, inheritance, coupling, the complexity of a dataset, and LOC of a particular software.

The software projects validated in this thesis include six open source java applications the PROMISE repository - ant, camel, ivy, jedit, log4j and prop. The brief description of these open source projects are as follows: Apache Ant (Another Neat Tool) is " a software tool that automates soft-ware build processes, such as compiling, running, testing, and assembling Java appli-cations" [68]. Apache Camel is " a message-oriented middleware framework to exchange, route and transform data using various protocols" [69]. Apache Ivy is " a dependency management tool used to manage (record, track, resolve, and report) project dependencies" [70]. jEdit is " a programmer's text editor with a number of features and plugins. Its powerful search engine for regular expressions, syntax high-lighting, and auto-indentation makes it particularly appealing to those working with Java and XML" [71]. Log4j is " a java based logging library popularly used by a variety of applications" [72]. Apache Props Antlib is " a library of supplementary handlers for Apache Ant properties resolution"[73].

The characteristics of defects of datasets of AEEEM[53], JIRA[54], PROMISE[52],

and Android[55] are given in Table 2.1, Table 2.2, Table 2.3 and Table 2.4 respectively.

Table 2.1: Characteristics of Datasets of AEEEM Repository

Dataset	Version	Total Components	Non-Buggy	Buggy	% of Buggy
Equinox (EQ)	3.4	324	195	129	40
JDT Core (JDT)	3.4	997	791	206	21
PDE UI (PDE)	3.4.1	1497	1288	209	14
Mylyn (MYL)	3.1	1862	1617	245	13
Apache Lucene (AL)	2.4.0	691	627	64	9

Table 2.2: Characteristics of Datasets of JIRA Repository

Dataset	Version	Total Components	Non- Buggy	Buggy	% of Buggy
Active MQ	5.0.0	1884	1591	293	16
Derby	10.5.1.1	2705	2322	383	14
Groovy	1.6 (Beta 1)	821	751	70	9
Hbase	0.94.0	1059	841	218	21
Hive	0.9.0	1416	1133	283	20
JRuby	1.1	731	644	87	12
Wicket	1.3.0 (Beta 2)	1763	1633	130	7

The study on defect categorization employed the defect data of five modules of Android software [74], that is available on GITHUB repository [55]. The changelogs between two versions in the repository have a description of the changes, that are used in finding the defects. The five Android modules and their versions between which change logs extracted were: Bluetooth (4.1-4.4), Browser (2.3-4.0), Calendar (4.1-4.4), Camera (2.3-4.0) and

MMS (2.3-4.0). The study used software called Defect Collection and Reporting System (DCRS)[75], that aids in extracting the bug reports from the changelogs between the two predetermined versions of Git repository. Each defect report extracted has the following data: a) Unique defect identifier; b) Source code file name, which has been affected by the defect correctness; c) the description of the defect; d) LOC added and LOC removed to fix a defect and e) the total of LOC added or removed for fixing the defect. The study applied text mining techniques to find keywords from the defect descriptions found in the bug reports generated by DCRS. Then the study performed preprocessing, feature selection using infogain measure and vector space model defined by Ruchika et al. [76][77] to obtain Top10, Top25, Top50 and Top100 ranked keywords. These keywords are used in building SDC models as independent features. The defects found from the bug reports are categorized based on maintenance effort and change impact and assign levels (low, medium and high) for every defect identified in the android application packages under this study. Table 2.4 lists out the count of different levels of defects in these datasets.

Table 2.3: Characteristics of Datasets of PROMISE Repository

Dataset	Version	Total Components	Non-Buggy	Buggy	% of Buggy
Ant	1.7	745	579	166	22
Camel	1.4	872	727	145	17
Ivy	2	352	312	40	11
Jedit	4	306	231	75	25
Log4j	1	135	101	34	25
Poi	2	314	277	37	12
Tomcat	6	858	781	77	9
Velocity	1.6	229	151	78	34

Table 2.3 continued from previous page

Dataset	Version	Total Components	Non-Buggy	Buggy	% of Buggy
Xalan	2.4	723	613	110	15
Xerces	1.3	453	384	69	15

Table 2.4: Count of Different Levels of Defects in Android Datasets

S.No.	Application	Versions	Maintenance Effort			Change Impact			Combination		
			Low	Medium	High	Low	Medium	High	Low	Medium	High
1	Bluetooth	4.1-4.4	26	27	26	15	54	10	27	27	25
2	Browser	2.3-4.0	196	191	199	328	110	148	200	193	193
3	Calendar	4.1-4.4	54	55	56	35	30	100	55	55	55
4	Camera	2.3-4.0	117	118	118	115	156	82	118	119	116
5	MMS	2.3-4.0	57	57	54	42	95	31	56	56	56

2.9 Model Development and Validation

The study employed Stratified K-Fold Cross-Validation technique with K=10, means that the datasets are divided into ten partitions or folds [78]. The training of the model is performed using nine folds, while the remaining fold is utilized for validation. This ten-fold process is repeated to ensure all folds are utilized for both training and validation purposes. The illustration of k-fold cross validations is given Figure 2.3. Stratified K-Fold Cross-Validation offers several advantages: Firstly, it mitigates bias arising from imbalanced class distributions, by ensuring the defective and non-defective instances are appropriately represented in training and validation sets. Secondly, by repeating the process ten times, we obtain robust estimates of model performance, reducing the impact of random variations and enhancing the reliability of the results. Moreover, this methodology allows for the assessment of model generalizability across different datasets, enhancing the applicability

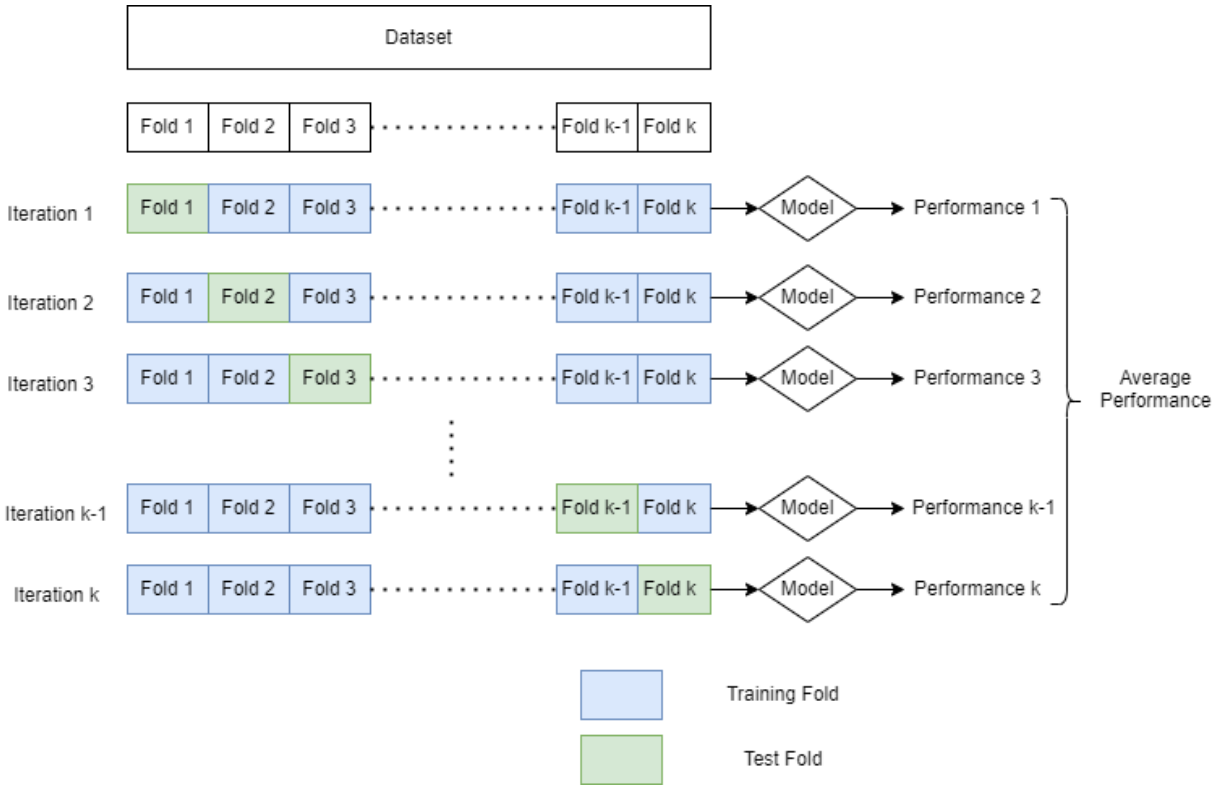

Figure 2.3: Illustration of k-Fold Cross Validation

of the developed SDP models.

2.10 Performance Measures

Defect prediction is a classification problem, where the model predicts whether the given class is a defective or not. The defective class is the positive class whereas the non-defective class is the negative class. The confusion matrix lists all four possible prediction outcomes in a binary classification problem as shown in Table 2.5. When a defective class

is classified as ”defective” correctly, it is considered a true positive (TP). Conversely, when a non-defective class is incorrectly classified as ”defective”, it is considered a false positive (FP). A true negative (TN) occurs when a non-defective class is correctly classified as non-defective, while a false negative (FN) occurs when a defective class is incorrectly classified as non-defective.

Table 2.5: Confusion Matrix

	Predicted Defective	Predicted Non-defective
Actual Defective	TP	FN
Actual Non-defective	FP	TN

Accuracy is the ratio of correctly predicted instances to the overall number of instances in a prediction model.

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN} \; x \; 100 \tag{2.26}$$

$$Error\ Rate = 1 - Accuracy \tag{2.27}$$

Precision is the proportion of true defective instances out of all the instances that are predicted to be defective.

$$Precision = \frac{TP}{TP + FP} \; x \; 100 \tag{2.28}$$

Recall or Sensitivity or True positive rate (TPR) is the ratio of correctly predicted defective

instances to the total number of actual defective instances.

$$TPR = \frac{TP}{TP + FN} \, x \, 100 \tag{2.29}$$

Specificity or Selectivity or True negative rate (TNR) is the proportion of correctly predicted non-defective instances to the total number of actual non-defective instances.

$$TNR = \frac{TN}{TN + FP} \, x \, 100 \tag{2.30}$$

False positive rate (FPR) is the proportion of non-defective instances that were incorrectly predicted as defective to the total number of actual non-defective instances.

$$FPR = \frac{FP}{TN + FP} \, x \, 100 \tag{2.31}$$

Accuracy measure is suitable for evaluating model performance on balanced datasets. However, in imbalanced datasets, the accuracy measure may be biased towards the majority class, leading to the failure of identifying the crucial minority class. To overcome this issue, several studies have suggested using performance metrics such as G-mean, Balance, Area under ROC (AUC) and Matthews Correlation Coefficient (MCC) to evaluate imbalanced data.

The G-mean (GM) is a metric that calculates the geometric mean of sensitivity (TPR) and specificity (TNR). It is useful for evaluating classification performance on imbalanced datasets because it takes into account the performance on both the majority and minority classes. "A low G-mean score indicates poor performance in correctly classifying the positive cases, even if the negative cases are correctly classified. [79]" Therefore, a high

G-mean score indicates a balanced performance across both classes.

$$GM = \sqrt{TPR * TNR} \tag{2.32}$$

Balance measure (BL) is Euclidean distance between a pair of (TPR, FPR) to that of an optimal value pair of TPR =1 and FPR = 0. A high BL indicates that TPR and FPR are close to their optimal values of 1 and 0 respectively.

$$BL = 1 - \sqrt{\frac{(1 - TPR)^2 + (0 - FPR)^2}{2}} \tag{2.33}$$

The area under the receiver operating characteristic (ROC) curve is commonly known as AUC. The ROC curve is plotted using the false positive rate (FPR) as the x-coordinate and the true positive rate (TPR) as the y-coordinate. The AUC represents the probability that a randomly selected defective instance will be ranked higher than a randomly selected non-defective instance [80]. AUC is often employed as it is less sensitive to imbalanced data. Higher AUC values indicate better performance of the machine learning classifier.

$$AUC = \int_0^1 TPR(FPR)d(FPR) \tag{2.34}$$

The Matthews Correlation Coefficient (MCC) is a metric used to evaluate the quality of binary classifications. It takes into account all four quadrants of a confusion matrix: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). The MCC is particularly useful for imbalanced datasets because it provides a balanced measure that can be used even when the classes are of very different sizes. The MCC is defined as follows:

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TN} + \text{FN})}} \tag{2.35}$$

The MCC produces a value between -1 and +1. +1 indicates a perfect prediction, 0 indicates that the prediction is no better than random and -1 indicates a total disagreement between prediction and observation.

2.11 Statistical Analysis

To validate the significance of performance improvements across different models, statistical tests are employed. Two of the most commonly used non-parametric tests in machine learning for comparing multiple algorithms are the Friedman Test and Wilcoxon Signed Rank Test.

The Friedman test is used to compare more than two techniques across multiple datasets. It checks for significant differences in the rankings of techniques [81]. It is a non-parametric equivalent of the repeated measures ANOVA test. This test is particularly suited for situations where the same parameter is measured under various conditions on the same subject. The Friedman test calculates the rank of each technique across multiple datasets, and the average rank provides the mean rank for that specific technique.

$$\chi_F^2 = \frac{12N}{k(k+1)} \left(\sum_{j=1}^{k} R_j^2 - \frac{k(k+1)^2}{4} \right) \tag{2.36}$$

where N is the number of datasets, k is the number of techniques, R_j is the rank of j^{th} the algorithm on each dataset. If the p-value is below the significance level (i.e., α =0.05), the null hypothesis (that all techniques perform equally) is rejected. Thus, the Friedman

test allows for comparing multiple algorithms across different datasets without assuming normality.

The Wilcoxon Signed Rank Test is a non-parametric test used to compare two related samples. This statistical test is employed to compare two related samples or repeated measurements on a single sample. The study used it to evaluate pairs of techniques with the aim of examining the null hypothesis, which posits that there is no statistically significant difference in the performance between the techniques [81].

$$W = \min(\sum_{i} R_i^+, \sum_{i} R_i^-) \tag{2.37}$$

where R^+ and R^- are the ranks of the positive and negative differences between the two samples. If the p-value is below the threshold (i.e., α =0.05, it implies a significant difference between the algorithms.

Chapter 3

Systematic Literature Review

3.1 Introduction

Software quality prediction plays a pivotal role in ensuring the long-term success and sustainability of software projects. As systems evolve, they are subjected to numerous changes, including bug fixes, enhancements, and updates, which can inadvertently introduce defects or degrade the quality of the system [82]. In this context, the ability to forecast which parts of the system are likely to require changes or are prone to defects can help organizations allocate resources efficiently, prioritize testing efforts, and reduce the cost and time involved in maintaining software.

Predictive models for software quality have traditionally relied on statistical techniques and expert judgment. However, the increasing availability of large-scale datasets and the rapid advancements in machine learning have transformed the way these models are developed and validated. Modern predictive models leverage a wide range of features

and metrics derived from the software itself, such as code complexity, historical defect data, and process-related factors. The incorporation of machine learning techniques into this process allows for more accurate, scalable, and adaptive models, providing valuable insights into software quality at various stages of development and maintenance. The process of systematically studying and developing the software quality predictive models is essential for the continuous improvement of software engineering practices, which forms the basis for this Systematic Literature Review (SLR).

This chapter conducts a systematic literature review to synthesize the state of the art in software quality prediction, with a particular focus on machine learning-based methodologies. The goal is to provide a comprehensive and structured analysis of the current knowledge, practices, gaps, and future directions in this field. The scope of this review encompasses the metrics, algorithms, techniques, and datasets used by researchers and practitioners to build and evaluate predictive models aimed at identifying defect-prone and change-prone software classes or modules.

Research Questions Guiding the Review

The review is structured around several research questions (RQs), each designed to address specific aspects of software quality predictive modeling. These RQs form the core of the systematic review and guide the literature search, selection, and analysis processes. The research questions are:

RQ1: What are the most used metrics in identifying the change and defect-prone classes/modules?

RQ2: What kinds of datasets are being used by researchers in software defect and change prediction?

RQ3: Which machine learning algorithms are effective in identifying the change and defect-prone classes/modules?

RQ4: How the researchers addressed issues (like imbalanced data, high dimensionality, parameter tuning, multicollinearity) affecting the performance of software quality prediction models?

RQ5: What are techniques being used for validation of software quality prediction models?

RQ6: What are the performance measures used for the evaluation of software quality prediction models?

RQ7: What are the statistical tests used by the researchers?

Overview of the Systematic Literature Review Process

The systematic literature review follows a structured process, adhering to the guidelines established by Kitchenham and Charters [83] for conducting rigorous and transparent reviews in software engineering. The process begins with the formulation of the research questions and the creation of a detailed review protocol. This is followed by an extensive search of relevant literature across multiple databases, ensuring that the most up-to-date and impactful studies are included in the review.

Once the literature is collected, a careful selection process is employed to filter out studies that do not meet predefined inclusion and exclusion criteria. Studies are then subjected to a quality assessment to ensure that only high-quality, relevant research is included in the final analysis. The selected studies are systematically reviewed and categorized based on their contribution to answering the RQs.

The results of the review are presented in a structured format, with separate sections dedicated to each research question. This ensures clarity and allows for a focused analysis of each aspect of software quality predictive modeling. The review also includes a discussion section, which synthesizes the findings and highlights key trends, gaps, and future research directions.

The chapter is structured as follows: Section 3.2 outlines the review protocol, including the search strategy, inclusion/exclusion criteria, and quality assessment methods. Section 3.3 presents the review results, organized by research question. Section 3.4 provides a discussion of the findings, offering insights into the current state of research and identifying opportunities for future work.

3.2 Review Protocol

This section outlines the systematic approach adopted to conduct the systematic literature review (SLR) to ensure the transparency, replicability, and rigor of the research process. The protocol defines the research questions, search strategy, inclusion/exclusion criteria, and quality assessment criteria, which guide the selection and evaluation of studies relevant to answering the research questions. The objective of this protocol is to provide a comprehensive and structured analysis of the existing literature related to software defect and change prediction, with a focus on metrics, datasets, machine learning algorithms, validation methods, performance metrics, and statistical tests used in SQP models.

3.2.1 Search Strategy

The search strategy is critical to ensuring that the review captures all relevant studies published in the field. A comprehensive search will be performed across multiple databases to locate studies that address the research questions. The databases chosen for this review include:

- IEEE Xplore
- ACM Digital Library
- SpringerLink
- ScienceDirect
- Google Scholar
- Wiley Online Library

These databases were selected based on their relevance to the field of software engineering and machine learning, as they index a wide range of peer-reviewed conference papers, journal articles, and relevant technical reports.

Keywords and search terms are constructed to target studies related to SDP, change prediction, and machine learning in software quality. The search is based on combinations of key terms related to the research questions, such as: ”software defect prediction”, ”change prediction”, ”machine learning algorithms”, ”software metrics”, ”imbalanced data in software prediction”, ”high dimensionality in software models”, ”parameter tuning in

machine learning", "cross-project validation", "performance measures in software quality", ans "statistical tests in software defect prediction"

The search string is customized to fit the search query requirements of the databases. Boolean operators will be used to combine search terms and ensure relevant studies are retrieved. The search string formed is as follows:

("software" AND ("defect" OR "bug" OR "fault" OR "change" OR "effort" OR "maintenance" OR "quality")) AND ("machine learning" OR "support vector machine" OR "naïve bayes" OR "deep learning" OR "neural network" OR "ANN" OR "CNN" OR "RNN"OR "ensemble learning" OR "random forest" OR "decision tree" OR "CART") AND ("variables" OR "parameters" OR "metrics" OR "object oriented metrics" OR "OOM") AND ("validation" OR "empirical" OR "design" OR "development") AND (("imbalanced data" OR "data sampling" OR "undersampling" OR "oversampling" OR "SMOTE" OR "cost sensitive") OR ("hyperparameter tuning" OR "parameter tuning" OR "grid search" OR "random search") OR ("feature selection" OR "dimensionality reduction") OR ("multicollinearity")) OR ("evolutionary" OR "search" OR "optimized" OR "heuristic" OR "particle swarm" OR "harmony search" OR "simulated annealing" OR "bat search" OR "swarm intelligence" OR "firefly search" OR "gravitational serach" OR "inclined planes sytem" OR "bio-inspired" OR "genetic algorithm" OR "Grey Wolf" OR "cuckoo serach" OR "ant colony" OR "artificial bee colony") AND ("method" OR "technique" OR "algorithm" OR "variant" OR "model") OR ("cross-validation" OR "hold-out validation" OR "k-fold cross validation") OR ("performance measure" OR "performance metric" OR "accuracy" OR "AUC" OR "MCC" OR "gmean" OR "balance") OR ("statistically" OR "validated" OR "statistical" OR "statistical test" OR "paired test" OR "wilcoxon" OR "friedman" OR "ANOVA"))

The search will focus on primary studies published between January 2010 and July 2024 to ensure the review includes up-to-date research in software defect prediction and change-prone module identification.

3.2.2 Inclusion and Exclusion Criteria

Inclusion Criteria

This will be applied to filter studies based on their relevance and alignment with the RQs. To ensure the review focuses on high-quality and relevant studies, only those studies that meet the following criteria will be included:

- *Relevance to Software Quality Prediction:* Studies must specifically address software defect or change prediction using machine learning techniques.
- *Use of Empirical Data:* The study must present empirical results based on real-world datasets or simulation-based experiments related to software quality prediction.
- *Focus on Machine Learning Algorithms:* Studies should involve the application of machine learning algorithms (e.g., decision trees, support vector machines, neural networks, or ensemble methods) in software defect or change prediction.
- *Metrics, Datasets, and Algorithms:* Studies should explore or apply metrics, datasets, or machine learning algorithms relevant to identifying change or defect-prone modules.
- *Handling of Issues in Prediction Models:* Studies that address common issues such as imbalanced data, parameter tuning, high dimensionality, and multicollinearity will be included.

- *Studies Evaluating Model Performance:* Studies must evaluate the performance of the models using well-established measures such as accuracy, precision, recall, F1-score, AUC, etc.
- *Validation Techniques:* Studies employing validation methods (e.g., cross-project validation, k-fold cross-validation) to ensure the robustness of the predictive models will be included.
- *Language:* Only studies written in English will be included.
- *Peer-Reviewed Studies:* Only peer-reviewed journal articles, conference papers, and technical reports will be included to ensure the quality of the reviewed literature.

Exclusion Criteria

The exclusion criteria help eliminate studies that do not meet the necessary quality or focus on irrelevant topics. Studies will be excluded based on the following criteria:

- *Irrelevant Focus:* Studies that focus on software quality without applying machine learning techniques or predictive models for defect/change prediction.
- *Lack of Empirical Data:* Studies that provide theoretical discussions or surveys without presenting empirical data or results from experiments.
- *Non-Predictive Models:* Studies that focus on areas unrelated to prediction modeling, such as software development methodologies, software maintenance practices, or general software quality assurance.

- *Duplicate Studies:* Duplicate studies (e.g., the same study published in different venues) will be excluded, and only the most complete version of the study will be considered.
- *Non-English Studies:* Studies published in languages other than English will be excluded.
- *Non-Peer-Reviewed Sources:* Unreviewed sources such as preprints, editorials, blog posts, and non-peer-reviewed conference papers will be excluded.

3.2.3 Quality Assessment Criteria

Quality assessment is essential to ensure that the studies included in the review are reliable and of high academic value. The following quality assessment criteria will be applied to evaluate the studies:

- *Relevance to Research Questions:* The study's relevance to the research questions will be assessed. Does the study directly contribute to answering at least one of the formulated RQs?
- *Clarity of Research Objectives:* The study must clearly define its research objectives, hypotheses, or goals, particularly regarding the evaluation of machine learning algorithms for defect/change prediction.
- *Appropriateness of Methodology:* The study's methodology will be scrutinized to ensure it is rigorous, replicable, and appropriate for addressing the problem. Studies should clearly explain the datasets, machine learning algorithms, evaluation metrics, and statistical tests used.

- *Quality of Empirical Evidence:* The study must provide sufficient empirical evidence in the form of experimental data, case studies, or simulation results. The use of real-world software projects and datasets will be considered a positive indicator of quality.
- *Addressing Common Issues:* The study must discuss or address common challenges (e.g., imbalanced data, multicollinearity, overfitting) in machine learning-based predictive models.
- *Validation of Results:* The study should employ appropriate validation techniques (e.g., cross-validation, cross-project validation) to assess the robustness and generalizability of the models.
- *Statistical Rigor:* Studies that employ statistical tests to support their results will be favored. The use of statistical tests to demonstrate the significance of the findings (e.g., t-tests, ANOVA, Wilcoxon test) will be seen as a mark of high-quality research.
- *Clear Presentation of Results:* The clarity with which the study presents its findings, including the discussion of the results, comparisons with related work, and the implications for future research, will be evaluated.

Based on the defined search criteria, approximately 200 studies were initially identified, out of which 54 were selected as primary studies after a thorough evaluation against the established quality assessment criteria. Table 3.1 provides a detailed list of these selected studies.

Table 3.1: List of Primary Studies

Study Id	Study Name	Reference	Study Id	Study Name	Reference
PS1	Menzies (2010)	[84]	PS28	Manjula (2019)	[85]
PS2	Zheng (2010)	[86]	PS29	Tantithamthavorn (2018)	[87]
PS3	Song (2010)	[88]	PS30	Turabieh (2019)	[89]
PS4	Alsmadi (2011)	[90]	PS31	Shippey (2019)	[91]
PS5	Gao (2011)	[92]	PS32	Cai (2020)	[93]
PS6	Gao (2012)	[94]	PS33	Bal (2020)	[95]
PS7	Shivaji (2012)	[96]	PS34	Rhmann (2020)	[97]
PS8	Okutan 2014	[98]	PS35	Pandey (2020)	[18]
PS9	Wang (2013)	[99]	PS36	Deng (2020)	[100]
PS10	Liu (2014)	[101]	PS37	Alsghaier (2020)	[102]
PS11	Czibula (2014)	[103]	PS38	Yedida (2021)	[104]
PS12	Abaei (2015)	[105]	PS39	Ali (2021)	[106]
PS13	Erturk (2015)	[107]	PS40	Ulan (2021)	[108]
PS14	Arar (2015)	[66]	PS41	Zhao (2021)	[109]
PS15	Jin (2015)	[110]	PS42	Ardimento (2022)	[111]
PS16	Yang (2015)	[112]	PS43	Nevendra (2022)	[113]
PS17	Dôres (2016)	[114]	PS44	Manchala (2022)	[115]
PS18	Marian (2016)	[116]	PS45	Wang (2023)	[117]
PS19	Panichella (2016)	[118]	PS46	Yang (2023)	[119]
PS20	Rathore (2017)	[120]	PS47	Mafarja (2023)	[121]
PS21	Zhang (2016)	[122]	PS48	Ali (2023)	[123]
PS22	Huda (2017)	[124]	PS49	Yu (2023)	[125]
PS23	Ni (2017)	[126]	PS50	Khleel (2024)	[127]
PS24	Chen (2018)	[128]	PS51	Meher (2024)	[129]
PS25	Song (2018)	[130]	PS52	Ismail (2024)	[131]
PS26	Abaei (2020)	[132]	PS53	Garg (2024)	[133]
PS27	Chen (2018)	[134]	PS54	Wang (2024)	[135]

3.3 Review Results

The results extracted from the primary studies are presented in this section.

3.3.1 Results Specific to RQ1

Table 3.2: Metric Suites used in Software Quality Prediction

Metric Suite	Metrics
Chidamber & Kemerer (C&K)[21]	WMC (Weighted Methods per Class)
	DIT (Depth of Inheritance Tree)
	NOC (Number of Children)
	CBO (Coupling Between Objects)
	RFC (Response for Class)
	LCOM (Lack of Cohesion of Methods)
Li & Henry[22]	NOM (Number of Methods)
	NOC (Number of Children)
	NSM (Number of Static Methods)
	NSF (Number of Static Fields)
MOOD (Metrics for Object-Oriented Design) [24]	Method Hiding Factor (MHF)
	Attribute Hiding Factor (AHF)
	Method Inheritance Factor (MIF)
	Attribute Inheritance Factor (AIF)
	Coupling Factor (CF)
	Polymorphism Factor (PF)
McCabe[136]	Cyclomatic Complexity (CC)
Halstead[137]	Volume (V)
	Effort (E)
	Difficulty (D)
	Vocabulary (n)
	Length (N)
Bieman & Kang[25]	Cohesion (COH)
	Lack of Cohesion in Methods (LCOM)
Briand et al.[26]	Number of Couplings (NOC)
	Coupling Between Methods (CBM)
Martin[27]	Afferent Coupling (Ca)
	Efferent Coupling (Ce)
	Instability (I)
	Distance from the Main Sequence (D)
Lee[28]	Fan-in
	Fan-out
	Design Size in Classes (DSC)
	Number of Methods (NOM)

Table 3.2 continued from previous page

Metric Suite	Metrics
Bansiya & Davis[29]	Coupling Between Object Classes (CBO)
	Depth of Inheritance (DIT)
	Lack of Cohesion (LCOM)
Tang et al.[30]	Coupling Measures
	Cohesion Measures
	Complexity Measures

In software defect and change prediction studies, various metric suites have been widely used to identify change and defect-prone classes or modules. These metrics often originate from object-oriented programming (OOP) paradigms, focusing on code attributes that can indicate a higher likelihood of defects or maintenance efforts. A significant portion of the research has centered around well-established metric suites, including but not limited to the Chidamber and Kemerer (C&K) suite [21], Li and Henry metrics [22], MOOD[24] , McCabe [136], Halstead [137], Bieman and Kang [25], Briand et al. [26], Martin [27], Lee [28], Bansiya and Davis [29], and Tang et al. [30]. Each of these suites provides different perspectives on code quality, coupling, cohesion, and complexity, which are essential for predicting defect and change-prone areas in a software system. Table 3.2 provides a detailed list of metrics used by researchers in identifying the change and defect-prone classes/modules.

3.3.2 Results Specific to RQ2

In the realm of software quality prediction, the choice of datasets plays a crucial role in the effectiveness and reliability of predictive models. Researchers have leveraged various datasets from different domains, each with unique characteristics that contribute to the overall understanding of software quality prediction. NASA MDP and Apache projects are

the most commonly used datasets, providing rich sources of data for predicting defects and changes in software. Researchers have also employed datasets from telecommunications systems, open-source projects, and emerging domains, contributing to a broader understanding of software defect prediction across different environments. The use of these datasets has enabled researchers to develop, validate, and refine prediction models that can be applied in both industry and academia to enhance software quality and reliability. The distribution of studies according to the datasets employed are presented in Table 3.3 and Figure 3.1.

Table 3.3: Datasets used in studies

Dataset	Number of Studies	Percentage of Studies	List of Studies
NASA MDP	23	29.87	PS1, PS2, PS3, PS4, PS9, PS10, PS11, PS12, PS13, PS15, PS22, PS28, PS29, PS32, PS35, PS37, PS39, PS44, PS53, PS54
Apache Projects	21	27.27	PS7, PS8, PS17, PS19, PS20, PS25, PS29, PS30, PS31, PS33, PS36, PS38, PS41, PS42, PS43, PS44, PS45, PS47, PS48, PS49, PS51
LLTS	2	2.6	PS5, PS6
Bugzilla, Columba, etc.	3	3.9	PS27, PS40, PS52
Eclipse	5	6.49	PS21, PS29, PS31, PS49, PS51
AEEEM	3	3.9	PS23, PS25, PS49
Android	1	1.3	PS34
GNU Compiler Collection	1	1.3	PS21
ReLink	1	1.3	PS23
OpenOffice	2	2.6	PS21, PS51
Mozilla	3	3.9	PS21, PS40, PS52
PostgreSQL	2	2.6	PS27, PS40
ZooKeeper	1	1.3	PS42

Table 3.3 continued from previous page

Dataset	Number of Studies	Percentage of Studies	List of Studies
Others	9	11.69	PS14, PS16, PS24, PS26, PS46, PS48, PS50, PS52, PS34

3.3.3 Results Specific to RQ3

Identifying change and defect-prone classes/modules is crucial for maintaining software quality and ensuring efficient resource allocation during the software development lifecycle. The effectiveness of machine learning algorithms in this context has been a subject of extensive research. By analyzing a comprehensive range of studies, it becomes clear that several algorithms consistently demonstrate significant efficacy in predicting defect-prone areas of software systems.

The reviewed studies showcase a variety of machine learning techniques that cater to the diverse nature of software metrics and defect data. Table 3.4 provides the list of the machine learning techniques employed in the studies. Fig 3.2 depicts the bar chart of different machine learning techniques used in the studies and Figure 3.3 depicts the percentage of studies employed different types of machine learning techniques.

- *Naïve Bayes (NB):* This probabilistic classifier leverages Bayes' theorem and is particularly adept at handling categorical data. Its simplicity and interpretability make it a preferred choice for defect prediction, especially when historical defect data is available.

- *Support Vector Machine (SVM):* SVM excels at finding optimal hyperplanes for classification tasks. Its ability to effectively manage high-dimensional data is beneficial

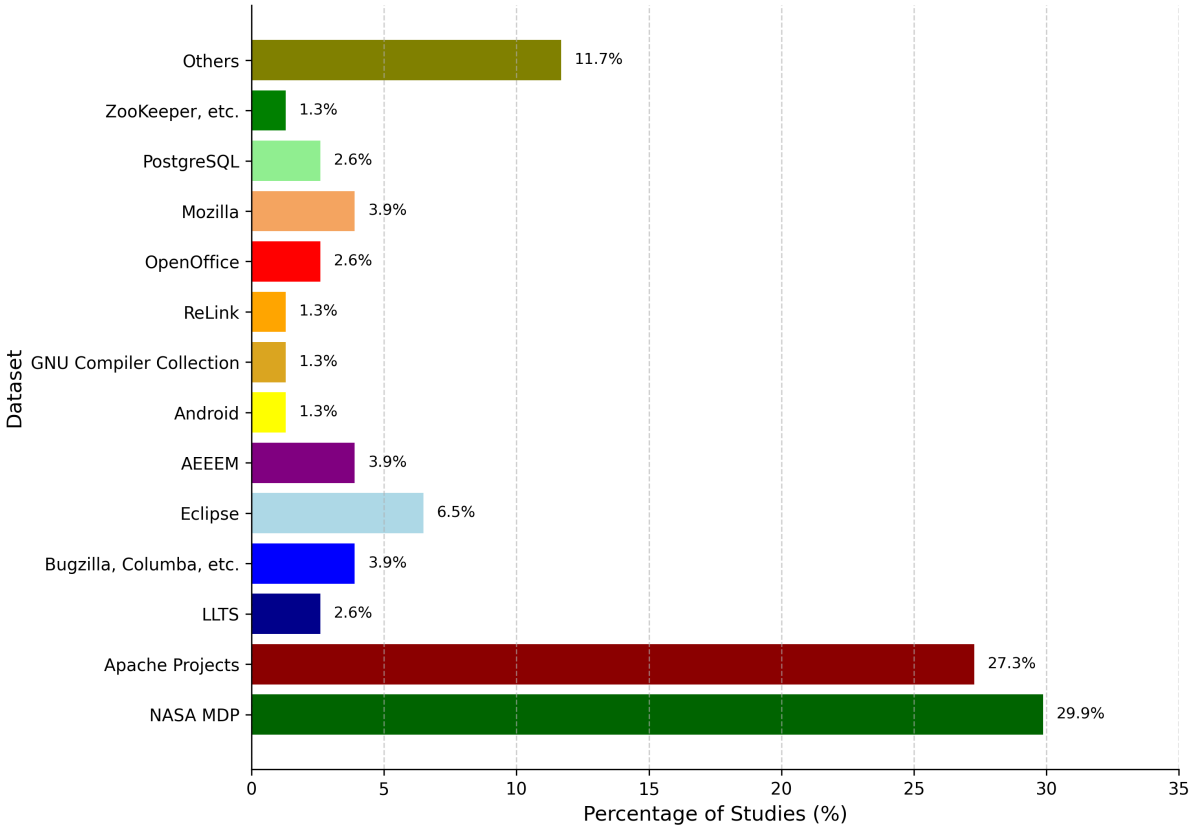

Figure 3.1: Percentage of studies using each dataset

for distinguishing between change-prone and defect-prone classes, often yielding high accuracy.

- *Random Forest (RF):* As an ensemble method, Random Forest integrates multiple decision trees to enhance prediction accuracy. Its robustness against overfitting makes it an effective tool for defect prediction, especially in diverse datasets.

- *Logistic Regression (LR):* This regression analysis technique is invaluable for binary

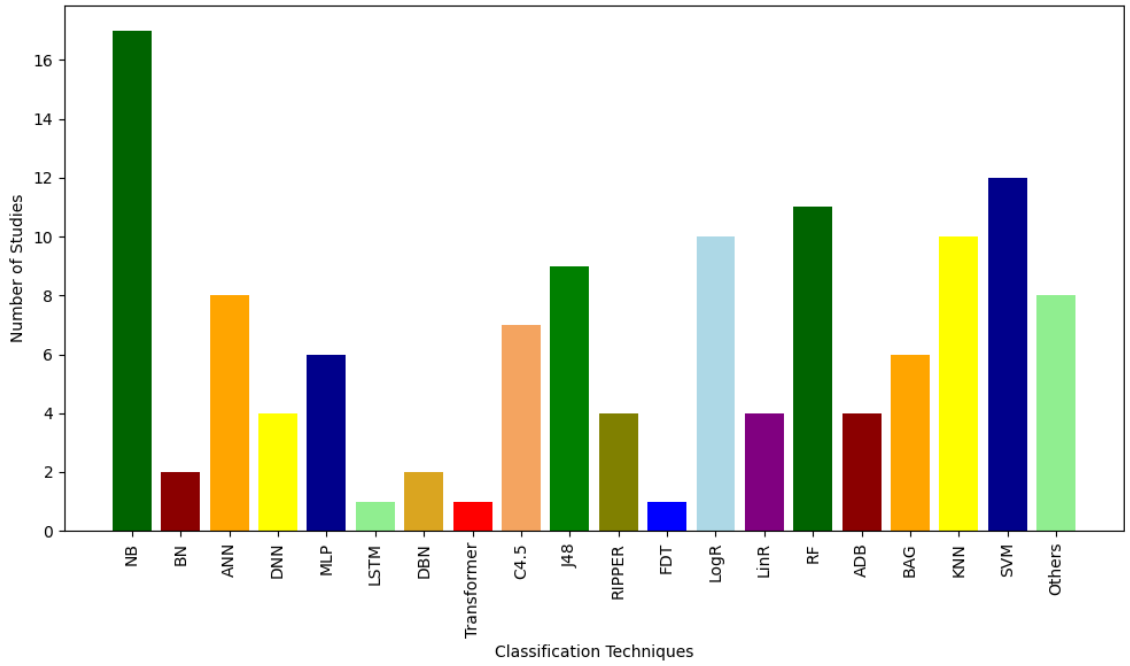

Figure 3.2: ML techniques used in the studies

classification tasks. It provides probabilistic interpretations of class memberships, facilitating the identification of defect-prone modules based on input features.

- *Artificial Neural Network (ANN):* ANNs are adept at modeling complex patterns within data. Their capacity for feature learning enables them to uncover underlying relationships that are crucial for predicting defect-prone areas in software systems.

- *Deep Learning:* Deep learning methods, including Deep Neural Networks (DNNs) and Long Short-Term Memory networks (LSTMs), have shown promise in defect prediction. They capture hierarchical data representations and temporal patterns, providing an advanced approach to modeling defect-prone areas.

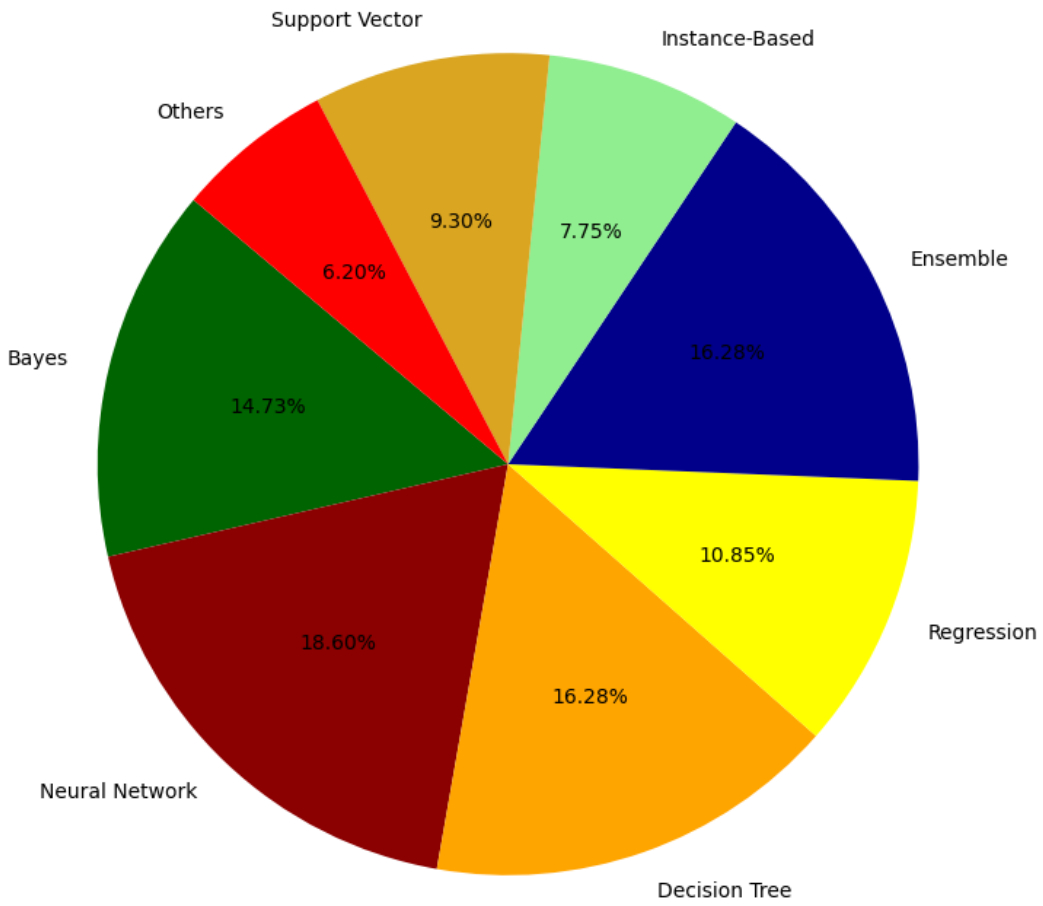

Figure 3.3: Distribution of studies by type of ML techniques used

- *Ensemble Techniques:* Ensemble methods combine predictions from multiple models, enhancing both robustness and accuracy in defect prediction tasks. Techniques like Random Forest, Bagging and AdaBoost improve generalization by reducing variance and bias.

Table 3.4: ML Techniques employed in the studies

Category	Classification Technique	No. of Studies	Studies
Bayes	Naïve Bayes (NB)	17	PS1, PS3, PS4, PS5, PS7, PS9, PS12, PS17, PS23, PS25, PS29, PS30, PS31, PS35, PS37, PS44, PS47
	Bayesian Network	2	PS8, PS41
Neural Network	Artificial Neural Network (ANN)	8	PS10, PS12, PS13, PS14, PS15, PS22, PS30, PS54
	Deep Neural Network (DNN)	4	PS28, PS44, PS48, PS53
	Multilayer Perceptron (MLP)	6	PS5, PS17, PS34, PS35, PS53
	Long Short Term Memory (LSTM)	1	PS36
	Deep Belief Network (DBN)	2	PS36, PS38
	Transformer	1	PS51
Decision Tree	C4.5	7	PS1, PS4, PS17, PS30, PS37, PS41, PS49
	J48	9	PS4, PS17, PS27, PS31, PS34, PS37, PS41, PS47
	RIPPER	4	PS1, PS25, PS35, PS49
	Fuzzy Decision Trees	1	PS18
Regression	Logistic Regression (LR)	10	PS1, PS5, PS23, PS25, PS30, PS39, PS43, PS44, PS47, PS49
	Linear Regression	4	PS19, PS20, PS27, PS43
Ensemble	Random Forest (RF)	11	PS9, PS12, PS17, PS23, PS24, PS25, PS27, PS31, PS34, PS41, PS53
	AdaBoost	4	PS2, PS17, PS24, PS27
	Bagging	6	PS1, PS11, PS27, PS29, PS35, PS41
Instance-Based	K-Nearest Neighbor (KNN)	10	PS1, PS5, PS6, PS17, PS21, PS30, PS35, PS37, PS44, PS47
Support Vector	Support Vector Machine (SVM)	12	PS4, PS5, PS6, PS7, PS13, PS17, PS22, PS32, PS37, PS44, PS47, PS53

Table 3.4 continued from previous page

Category	Classification Technique	No. of Studies	Studies
Others	Others	8	PS20, PS11, PS38, PS42, PS45, PS50, PS51, PS52

3.3.4 Results Specific to RQ4

The primary objective of software quality prediction models is to minimize software defects while maintaining high performance. However, the complexity of software systems introduces several challenges that hinder the prediction process. Among these, imbalanced data, high dimensionality, parameter tuning, and multicollinearity are commonly encountered issues that can negatively impact the modelâs performance. These challenges are often interrelated and must be addressed through careful preprocessing, optimization, and algorithm selection. To address these challenges, researchers have explored a variety of techniques aimed at improving the accuracy, robustness, and generalization of software quality prediction models. Table 3.5 presents overview of the methods researchers have employed to tackle these challenges effectively. Figure 3.4 depicts the number and percentage of studies out of the total primary studies have employed the methods to address the these challenges.

- Imbalanced data occurs when the number of instances in one class (e.g., defective modules) is significantly lower than the instances in the other class (e.g., non-defective modules). This can lead to biased predictions, where the model favors the majority class. Many studies have applied resampling techniques such as random undersampling (RUS), oversampling, SMOTE (Synthetic Minority Over-sampling

Technique), and cost-sensitive learning. These techniques aim to balance the class distribution, thereby reducing bias in predictions toward the majority class and improving the detection of minority (defective) instances.

- High dimensionality refers to datasets with a large number of features, which can result in overfitting, increased computational complexity, and difficulty in understanding the model's behavior. Not all features may contribute equally to the prediction task, and irrelevant or redundant features may degrade the model's performance. Feature selection methods such as filter methods (e.g., chi-square, information gain, and gain ratio) and wrapper-based techniques (e.g., genetic algorithm and binary ant colony optimization) were adopted to handle high-dimensional data (PS5, PS41). Studies like PS8 and PS14 used feature selection algorithms such as CFS (Correlation-based Feature Selection), ReliefF, and Cost-Sensitive Variance Score to reduce the feature set and eliminate irrelevant or redundant features. Reducing the number of features helps in lowering model complexity, improving interpretability, and preventing overfitting.

- Parameter tuning is essential for optimizing machine learning algorithms. Many algorithms rely on hyperparameters that must be fine-tuned to achieve the best results. Without proper tuning, models can either underperform or overfit the training data, limiting their generalization to unseen data. Researchers used several hyperparameter optimization techniques like Random Search (PS43), Grid Search (PS43), and metaheuristic algorithms such as genetic algorithms (GA) and particle swarm optimization (PSO) (PS37). In PS27, a NSGA-II (Non-dominated Sorting Genetic Algorithm II) was used for multi-objective hyperparameter tuning. Proper

tuning of hyperparameters improves the model's generalization ability and ensures optimal performance across different datasets.

- Multicollinearity arises when features in the dataset are highly correlated with each other, leading to instability in the model's predictions. This can cause issues with the interpretation of the model's output and reduce its accuracy. Although there is no direct mention of handling multicollinearity in the provided studies, feature selection techniques like correlation-based methods (PS14) can indirectly address multicollinearity by selecting features that are less correlated with each other. This reduces redundancy in the features, making the prediction model more stable and less sensitive to correlated inputs.

Table 3.5: Challenges addressed in the studies

Challenge	**No. of Studies**	**Studies**	**Techniques Used**
Imbalance Data	7	PS2, PS6, PS9, PS14, PS25, PS47, PS50	Cost-sensitive learning, SelectRUSBoost, Random undersampling, SMOTE, Cost-sensitive ANN, Weighted Average Centroid based Imbalance Learning Approach
Hyperparameter Tuning	5	PS14, PS21, PS27, PS29, PS43	ABC algorithm, REP Topic, NSGA-II, Caret R package, Random Search, Grid Search

Table 3.5 continued from previous page

Challenge	No. of Studies	Studies	Techniques Used
Feature Selection	6	PS5, PS8, PS10, PS14, PS41, PS53	Chi-square (CS) method, Information Gain (IG), Gain Ratio (GR), Symmetrical Uncertainty (SU), Kolmogorov-Smirnov Class Correlation-Based Filter (KS), Exhaustive Search (ES), Heuristic Search (HS), Automatic Hybrid Search (AHS), Correlation-based Feature Selection (CFS), Relief, Cost-Sensitive Variance Score, PCA, Fuzzy C-means Clustering Method

3.3.5 Results Specific to RQ5

Validation is a critical step in the development of software quality prediction models, ensuring that the models can generalize well to unseen data. A properly validated model gives confidence in its predictions and helps avoid overfitting or underfitting, which are common issues when dealing with machine learning models. Various validation techniques are employed to evaluate the performance of software quality prediction models, with the most common being cross-validation, which divides the dataset into multiple subsets for training and testing purposes. Among the different types of cross-validation, 10-fold cross-validation is the most frequently used in studies related to software quality prediction models. However, several other techniques like MxN-way cross-validation, leave-one-out cross-validation, and stratified sampling are also explored in specific cases. Figure 3.5 depicts the percentage of validation measured used across different studies.

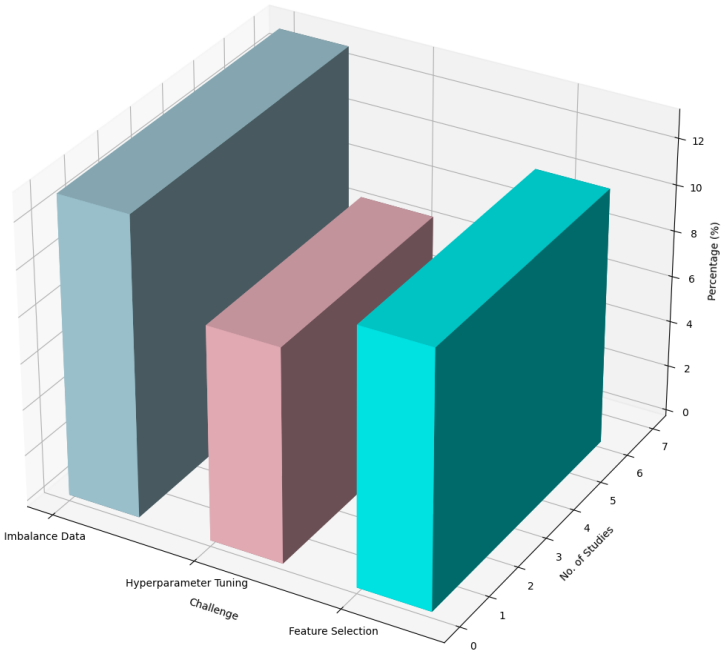

Figure 3.4: Number and percentage of studies employed techniques to address various challenges

- *10-Fold Cross-Validation:* In the 89% of the studies, 10-fold cross-validation is utilized as the primary validation technique. This technique divides the dataset into 10 equally sized subsets (folds). The model is trained on 9 folds and tested on the remaining one, and this process is repeated 10 times, with each fold being used as the test set once. The final performance metric is the mean of the 10 test results. 10-fold cross-validation provides a balanced trade-off between bias and variance. It reduces the risk of overfitting while still allowing enough data for the training process. It is computationally efficient compared to more exhaustive methods like leave-one-out

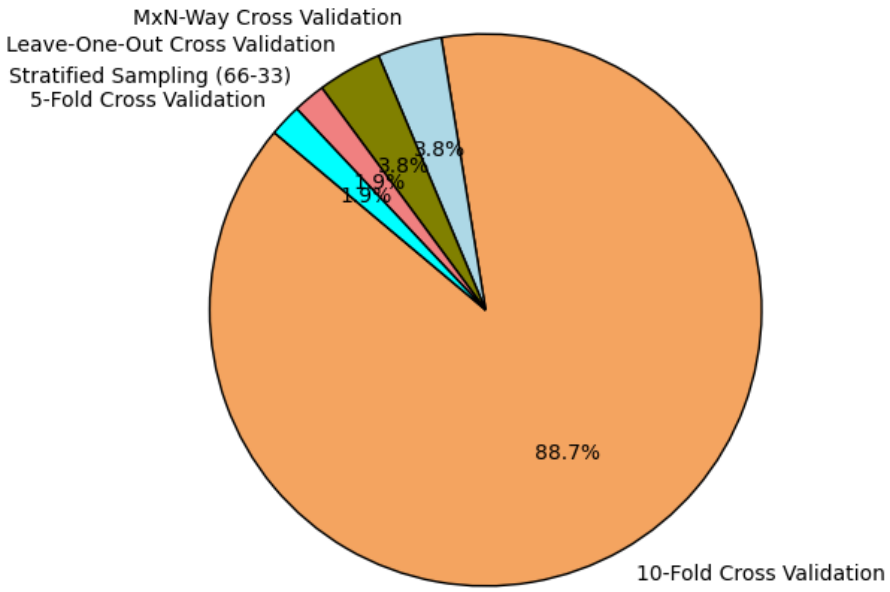

Figure 3.5: Usage of Validation Techniques Across Studies

cross-validation. Due to its stability and efficiency, 10-fold cross-validation has become a standard approach in machine learning research, including software quality prediction, and is the default validation technique in most software quality prediction studies.

- *MxN-Way Cross-Validation:* In some studies (PS3 and PS25), MxN-way cross-validation is employed. This method is an extension of basic cross-validation, where the dataset is divided into M subsets, and the validation process is repeated N times. This method adds an additional layer of randomness, as the division of the dataset is done multiple times, resulting in more robust validation. The N repetitions reduce

the potential bias introduced by any particular data split. While it offers a more comprehensive evaluation, it is more computationally expensive, which might limit its use when large datasets or complex models are involved.

- *Leave-One-Out Cross-Validation (LOO-CV):* Leave-one-out cross-validation (LOO-CV), used in studies like PS11 and PS17, involves training the model on the entire dataset except for one instance, which is used for testing. This process is repeated for every instance in the dataset. LOO-CV is highly exhaustive, as it evaluates the model on every single data point. This provides a complete understanding of how well the model generalizes. Despite its precision, LOO-CV is computationally expensive, especially for large datasets, as the model must be retrained as many times as there are instances in the dataset. Since the test set contains only one data point, there can be a high variance in the results across different test instances.

- *Stratified Sampling (66-33):* Stratified sampling, employed in PS12, involves splitting the dataset into a training set (66%) and a test set (33%) while ensuring that the proportion of instances in different classes remains the same across the training and test sets. Stratified sampling is especially important when the dataset is imbalanced, as it ensures that each fold contains a similar proportion of instances from each class. This method is simpler than cross-validation and requires only one split of the dataset. It is computationally efficient but may not provide as robust an evaluation as techniques like cross-validation.

- *5-Fold Cross-Validation:* In PS13, 5-fold cross-validation is used, which follows the same procedure as 10-fold cross-validation but divides the dataset into 5 parts instead of 10. 5-fold cross-validation is less computationally demanding than 10-fold, while

still offering a good measure of model generalization. However, with fewer folds, there is a higher risk of variance in the results.

3.3.6 Results Specific to RQ6

In software quality prediction research, the performance of machine learning models is typically assessed using a range of performance metrics. These metrics help to evaluate the predictive capability, robustness, and generalizability of models. Table 5.6 and Figure 5.6 presents a summary of performance measures extracted from various studies and the frequency with which they are used

- *AUC (Area Under the Curve):* The AUC is a widely used measure for evaluating the performance of classification models, particularly in imbalanced datasets. It measures the area under the receiver operating characteristic (ROC) curve, providing an aggregated measure of performance across all classification thresholds. The higher the AUC, the better the model's ability to distinguish between defect-prone and non-defect-prone classes. A total of 19 studies (e.g., PS1, PS5, PS9) used AUC as a primary metric.

- *F-measure/F1 Score:* The F1 score is a harmonic mean of Precision and Recall, offering a single metric that balances the trade-off between these two. It is especially useful when the dataset has an uneven class distribution, as it provides a more nuanced view of a model's performance. This metric was used in 17 studies (e.g., PS5, PS35, PS40), indicating its importance in software quality prediction.

- *Precision:* Precision, which measures the ratio of true positives among all predicted

positives, is crucial for models where false positives are costly. Thirteen studies (e.g., PS21, PS28, PS31) used Precision as a key metric, often in conjunction with Recall to give a fuller picture of performance.

- *Recall:* Also known as Sensitivity, Recall is the ratio of actual defect-prone classes correctly identified by the model. It is vital in applications where missing a defect-prone class is more costly than wrongly predicting a class as defect-prone. Recall was evaluated in 12 studies (e.g., PS29, PS50, PS54).

- *Accuracy:* Accuracy measures the ratio of correct predictions out of all predictions made. While it is a simple and intuitive metric, it is less useful in imbalanced datasets. Accuracy was still utilized in 10 studies (e.g., PS39, PS50, PS27), often alongside other metrics like AUC.

- *MCC (Matthews Correlation Coefficient):* MCC is a balanced measure of performance that takes into account true and false positives and negatives. It is particularly useful in situations with imbalanced classes. Six studies (e.g., PS31, PS41, PS50) utilized MCC.

- *G-mean:* G-mean is the geometric mean of the true positive rate and the true negative rate, used to assess a model's performance when the data is imbalanced. This metric was used in five studies (e.g., PS9, PS32), underscoring its importance in evaluating software quality prediction models in the face of skewed datasets.

- *Probability of Detection (PD) and Probability of False Alarm (PF):* PD is the rate of correctly identifying defect-prone classes, while PF is the rate of falsely predicting a defect-prone class. Both metrics are crucial in understanding the trade-offs in

prediction models. Five studies (e.g., PS9, PS24) reported PD and PF, highlighting their significance in quality prediction.

The analysis shows that AUC, F-measure, Precision, Recall, and Accuracy are the most commonly used performance measures in evaluating software quality prediction models. These measures provide a comprehensive view of the model's ability to handle both balanced and imbalanced datasets. Measures like MCC and G-mean are particularly useful in imbalanced scenarios, while Accuracy, though common, should be interpreted cautiously in such cases. The diversity of metrics used across the studies highlights the need for a multi-metric evaluation approach to fully capture the effectiveness of prediction models.

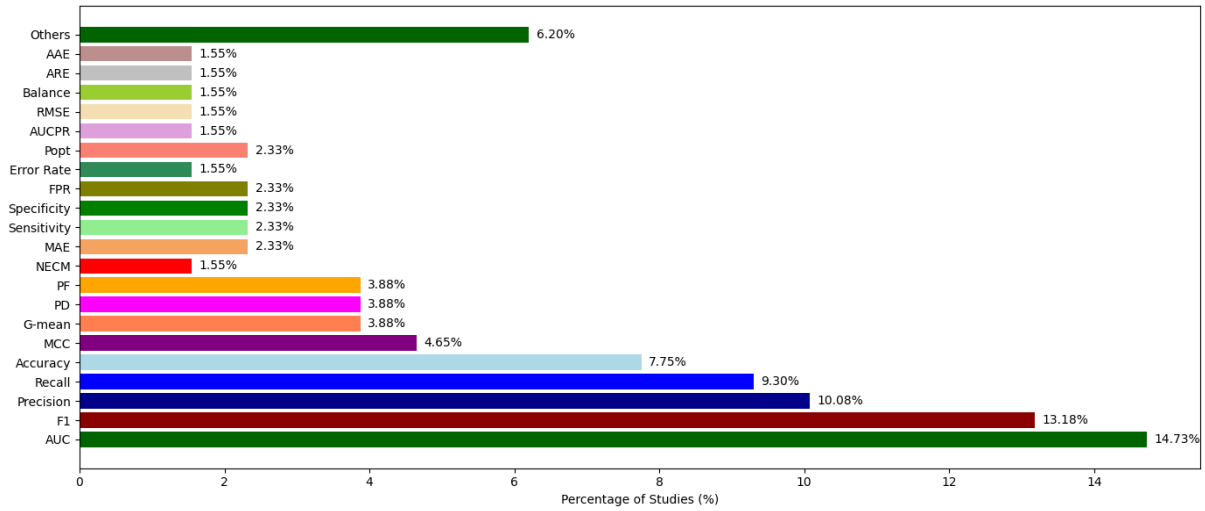

Figure 3.6: Usage of Performance Metrics across studies

Table 3.6: Performance Metrics employed in studies

Performance Measure	Count of Studies	List of Studies
AUC (Area Under the Curve)	19	PS1, PS4, PS5, PS8, PS9, PS11, PS13, PS14, PS15, PS23, PS24, PS29, PS30, PS35, PS38, PS41, PS44, PS47, PS50
F-measure / F1-score	17	PS5, PS7, PS21, PS28, PS29, PS31, PS35, PS36, PS37, PS40, PS41, PS42, PS44, PS45, PS50, PS51, PS54
Precision	13	PS4, PS21, PS28, PS29, PS31, PS37, PS39, PS40, PS49, PS50, PS54
Recall	12	PS4, PS21, PS28, PS29, PS31, PS37, PS39, PS40, PS44, PS50, PS54
Accuracy	10	PS10, PS14, PS28, PS29, PS37, PS39, PS48, PS50, PS54
MCC (Matthews Correlation Coefficient)	6	PS4, PS31, PS35, PS41, PS50
G-mean	5	PS9, PS24, PS29, PS32, PS44
PD (Probability of Detection)	5	PS9, PS11, PS14, PS24, PS32
PF (Probability of False Alarm)	5	PS4, PS9, PS14, PS24, PS32
NECM (Normalized Expected Cost of Misclassification)	2	PS2, PS14
MAE (Mean Absolute Error)	3	PS39, PS43, PS48
Sensitivity	3	PS4, PS10, PS28
Specificity	3	PS4, PS28, PS32
FPR (False Positive Rate)	3	PS12, PS28, PS53
Error Rate	2	PS12, PS37
Popt	3	PS27, PS40, PS52
Precision-Recall Curve (AUCPR)	2	PS35, PS50
RMSE (Root Mean Square Error)	2	PS39, PS50
Balance	2	PS9, PS14
ARE (Average Relative Error)	2	PS20, PS33
AAE (Average Absolute Error)	2	PS20, PS33

Table 3.6 continued from previous page

Performance Measure	Count of Studies	List of Studies
Others	1 (per measure)	PS45 (AFPrecision, AFRecall), PS43 (MAP, MRR), PS49 (IFA, PMI, PLI, PofB, PofB/PMI, PofB/PLI), PS52 (F0.5-score, Benefit)

3.3.7 Results Specific to RQ7

This section answers the statistical tests utilized in the studies reviewed, emphasizing their application and relevance in analyzing data. Table 1.7 and Figure 1.7 summarizes the statistical tests employed in the the studies.

The analysis reveals a diverse range of statistical tests employed by researchers in the reviewed studies. The Wilcoxon signed-rank test and t-test were the most frequently used, indicating a preference for methods suitable for small sample sizes or non-normally distributed data. The presence of non-parametric tests like the Mann-Whitney U test, Kruskal-Wallis test, and Friedman test suggests a recognition of the limitations of parametric assumptions in many datasets. Additionally, the inclusion of effect size measures like Cliff's δ emphasizes the importance of understanding not just the statistical significance but also the practical significance of research findings.

These tests were selected based on the nature of the data and the specific research questions addressed, highlighting the critical role of appropriate statistical methods in deriving meaningful conclusions from research studies. This variety reflects the complexity of data analysis in research, necessitating the careful selection of statistical tools to suit specific data characteristics and study designs.

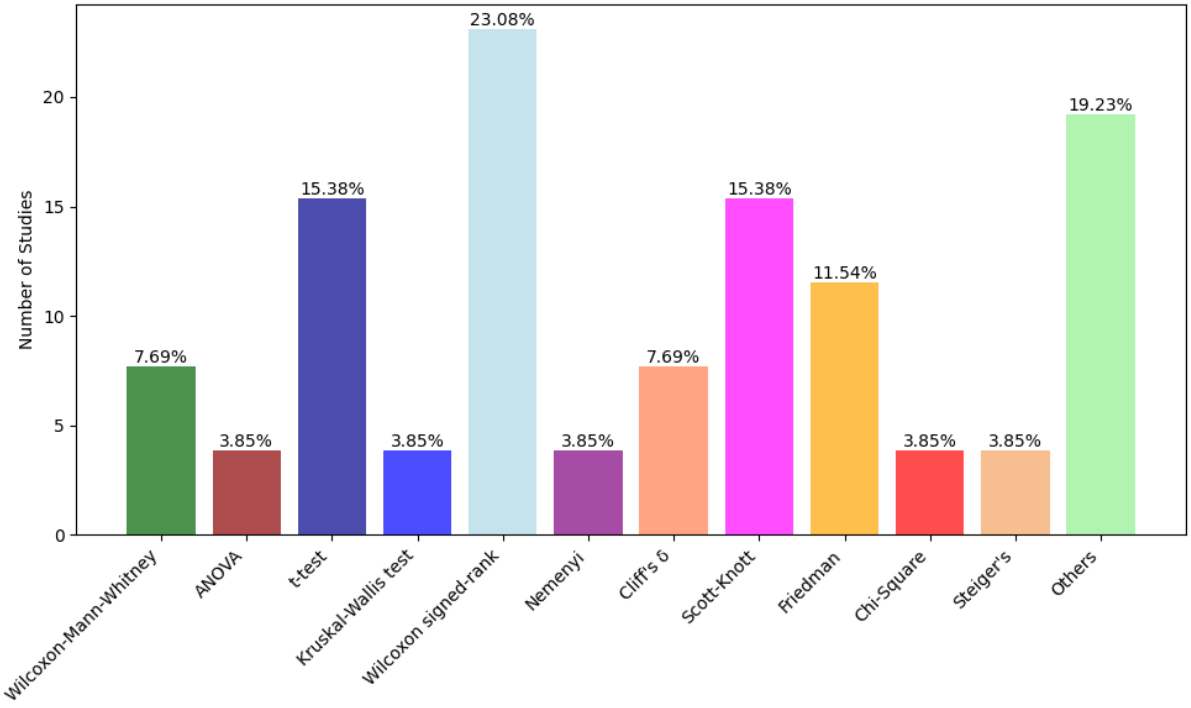

Figure 3.7: Statistical tests conducted in studies

Table 3.7: Statistical tests conducted by studies

Statistical Test	No. of Studies	List of Studies
Wilcoxon signed-rank test	6	PS21, PS23, PS30, PS37, PS44, PS47
t-test	4	PS8, PS9, PS21, PS23
Scott-Knott test	4	PS27, PS38, PS41, PS49
Friedman test	3	PS33, PS34, PS43
Wilcoxon-Mann-Whitney test	2	PS1, PS35
Cliff's δ	2	PS27, PS31
ANOVA	1	PS5
Kruskal-Wallis test	1	PS20
Nemenyi test	1	PS24
Chi-Square	1	PS34
Steiger's test	1	PS40

3.4 Discussion

The literature review serves as a critical foundation for understanding the current state of research in software defect and change prediction. This review aims to synthesize the findings from various studies, highlighting key insights, identifying gaps, and establishing a clear motivation for further research in this domain. The following subsections outline the outcomes of the literature review, the gaps identified, and the motivation behind this research.

Outcomes of Literature Review

The review of existing literature reveals several notable outcomes:

- *Metrics Utilization:* A wide variety of metrics have been employed in the identification of change and defect-prone classes/modules, with Object-Oriented (OO) metrics being predominant. Researchers have utilized metrics such as cyclomatic complexity, coupling, cohesion, and code churn to gauge the quality of software modules.

- *Datasets Analysis:* A diverse set of datasets has been used across studies, with notable examples including the NASA MDP and PROMISE repositories. However, the reliance on specific datasets raises concerns about the generalizability of the findings.

- *Machine Learning Techniques:* Several ML algorithms have been applied to the defect prediction problem, with notable mentions including Naïve Bayes, Support

Vector Machines (SVM), Random Forests, and Neural Networks. Each algorithm's performance has been evaluated, showcasing a range of effectiveness in predicting defect-prone modules.

- *Validation Techniques:* Various techniques have been employed for validating software quality prediction models, including cross-validation, bootstrapping, and hold-out methods, ensuring that the models developed are robust and reliable.

- *Performance Measures:* A plethora of performance measures have been utilized to evaluate the effectiveness of prediction models, including accuracy, precision, recall, F1-score, and area under the curve (AUC). These measures provide a comprehensive view of model performance.

- *Statistical Tests:* Researchers have employed various statistical tests to validate the significance of their findings, such as t-tests, ANOVA, and chi-squared tests, enhancing the reliability of their results.

Identified Gaps

Despite the substantial body of work in the field, several gaps remain that necessitate further investigation:

- *Need for Categorization of Defects:* While many studies focus on predicting the existence of defects, there is a pressing need for categorization of defects based on their impact, severity, and nature. This categorization could facilitate targeted interventions and improve overall software quality.

- *Metrics for New Programming Paradigms:* The rapid evolution of programming paradigms, including Agile, DevOps, and microservices, necessitates the development and utilization of new metrics. Most studies have relied heavily on traditional OO metrics, which may not adequately capture the complexities introduced by these modern paradigms.

- *Factors Affecting Performance:* Various factors affecting the performance of software quality prediction models, such as imbalanced data, high dimensionality, parameter tuning, and multicollinearity, have been identified. However, a limited percentage of studies have incorporated these factors into their models, leading to potentially skewed results.

- *Limited Dataset Diversity:* The majority of studies have relied on a small subset of datasets, raising concerns about the generalizability of the results. A broader range of datasets should be explored to validate findings across different contexts and environments.

- *Integration of Advanced Techniques:* The integration of advanced machine learning techniques, such as ensemble methods, deep learning, and hybrid models, remains underexplored. These approaches could offer significant improvements in predictive performance and robustness.

- *Lack of Comprehensive Evaluation Frameworks:* There is a need for comprehensive evaluation frameworks that integrate performance measures, validation techniques, and statistical tests, providing a holistic view of model effectiveness.

In summary, the outcomes of the literature review underscore the importance of

evolving research in software defect and change prediction, while the identified gaps highlight areas ripe for further exploration. This research aims to fill these gaps and pave the way for future advancements in the field.

Chapter 4

Categorization of Software Defects based on Maintenance Effort and Change Impact

4.1 Introduction

Software maintenance is a crucial phase within the software development lifecycle. This phase encompasses tasks such as addressing defects that emerge during production and integrating new customer requirements and change requests. Given the real-time nature of these changes in the production environment, there is a pressing need for their swift implementation and testing, all while working within the constraints of limited resources [138] [139]. Software Defect Categorization (SDC) emerges as a valuable strategy to efficiently manage these resources during the maintenance phase. It involves assigning a

defect level (high, medium, or low) based on specific defect attributes like criticality and priority [140]. Existing research in defect categorization has demonstrated that basing the defect level on maintenance effort and change impact yields robust predictive capabilities compared to other defect attributes. The study proposed in [141] explored an approach by combining maintenance effort and change impact to construct defect categorization models. Maintenance effort, in this context, refers to the estimation of effort needed to rectify a defect, often quantified by the number of lines of code (LOC) that require modification (addition, deletion, or alteration). Change impact, on the other hand, gauges the number of classes that need modification to resolve a defect. Defects categorized as high level based on maintenance effort demand more resources for their resolution, while those categorized as high level based on change impact necessitate additional resources for regression testing. Defects categorized as high level based on both attributes require substantial resources for both defect removal and regression testing [142].

This chapter focuses on three key experiments aimed at exploring different machine learning techniques for categorizing software defects based on maintenance effort and change impact:

4.2 Method

The research methodology of this chapter is presented in Figure 4.1.

4.2.1 Datasets

The study analyzes the defect data of five modules of Android software [74], that is available on GITHUB repository [55]. The details of this dataset is presented in Section

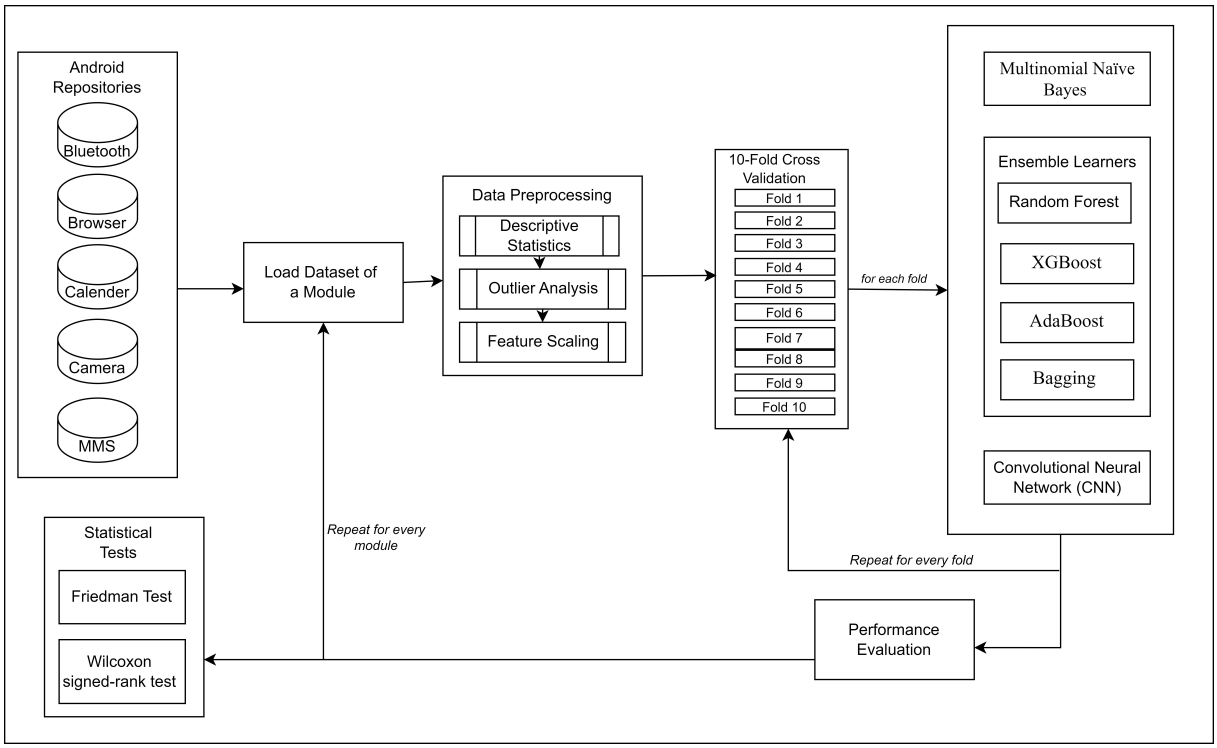

Figure 4.1: Research Methodology of Proposed Approach

2.8 of Chapter 2 and their characteristics are given in Table 2.4.

4.2.2 Classifiers

In this chapter, three main approaches are employed for Software Defect Categorization (SDC). Each study utilizes different classifiers to analyze and categorize software defects based on distinct algorithmic techniques.

- *Multinomial Naïve Bayes (MNB):* The first study applied the Multinomial Naïve Bayes algorithm to develop SDC models. MNB is known for its effectiveness in text

categorization and was adapted here to categorize software defects.

- *Ensemble Learning Techniques:* The second study used ensemble learning methods to enhance the robustness of the SDC models. Four prominent ensemble techniques were employed - Random Forest, XGBoost, AdaBoost, and Bagging - to evaluate their performance in software defect categorization.

- *Convolutional Neural Networks (CNNs):* In the third study, Convolutional Neural Networks, were utilized to construct SDC models, leveraging the strengths of CNNs in handling complex data representations for defect categorization.

4.2.3 Validation and Performance Evaluation

In this chapter, 10-fold cross-validation [78] was employed to ensure the robustness of the developed Software Defect Categorization (SDC) models. This technique divides the dataset into 10 subsets, using 9 subsets for training and the remaining one for testing, and the process is repeated 10 times as illustrated in Figure 2.3. This method helps to mitigate the risks of overfitting and provides a more reliable estimate of the model's performance.

The performance of the models was evaluated using the Area Under the Curve (AUC) metric as given at equation 2.18, which assesses the model's ability to distinguish between defect categories. AUC is a widely accepted measure for classification tasks and is crucial for comparing the effectiveness of different classifiers.

Additionally, statistical tests were performed to validate the significance of the results. The Friedman test was used to detect any statistically significant differences between the approaches as given at equation 2.20, while the Wilcoxon signed-rank test was employed as a post-hoc analysis to perform pairwise comparisons between approaches as given at

equation 2.21. These tests ensured that the observed differences in model performance were not due to random variation.

4.3 SDC with Multinomial Naïve Bayes (NBM) Algorithm

Firs, the study builds the Software Defect Categorization (SDC) models using NBM classification algorithm [11]. Three SDC models (low level, medium level, and high level) using NBM classification algorithm are developed for each dataset under study.

A naïve bayes classifier assumes that each of the variables it uses are conditionally independent of one another given some class. More formally, to calculate the probability of observing features f_1 through f_n, given some class c, under the naïve bayes assumption the following holds:

$$p(f_1, .., f_n/c) = \prod_{i=1}^{n} p(f_i/c) \tag{4.1}$$

This means that when we want to use a Naïve Bayes classifier to classify a new example, the posterior probability is much simpler to work with:

$$p\left(c/f_1, .., f_n\right) \;\; \alpha \;\; p(c)p(f_1/c)...p(f_n/c) \tag{4.2}$$

These assumptions of independence are rarely true, which may explain why some have referred to the model as the ”Idiot Bayes” model, but in practice, Naïve Bayes models have performed surprisingly well, even on complex tasks where it is clear that the strong independence assumptions are false. The term Multinomial Naïve Bayes means that each $p(f_i/c)$ is a multinomial distribution and holds good for ordinal values. In summary, NBM

model is a special case of Naïve Bayes model where a multinomial distribution is used for each of its features.

4.3.1 Experimental Results

A. SDC Model using Maintenance Effort

The AUC values of the SDC models developed in accordance with Maintenance effort using the NBM classification technique on the five datasets used in the study are shown in the Table 4.2. The developed SDC models identify three specific levels allocated to software defects i.e. 'low', 'medium' or 'high'. The study developed four models at each level using a different number of predictor variables (Top 10, Top 25, Top 50 and Top 100). All the AUC values which are greater than 0.7 are marked in bold. It is evident from the Table 4.2 that the ranges of AUC values of the 'low level' SDC models of Bluetooth, Browser, Calendar, Camera and MMS datasets ranged from 0.753-0.913, 0.585-0.653, 0.569-0.837, 0.610-0.826 and 0.421-0.834 respectively. The ranges of AUC values obtained by 'medium level' SDC models were 0.750-0.891 (Bluetooth), 0.574-0.669 (Browser), 0.482-0.788 (Calendar), 0.626-0.815 (Camera) and 0.448-0.726 (MMS). Similarly, the ranges of AUC value for 'high level' SDC models 0.687-0.881, 0.540-0.604, 0.621-0.900, 0.713- 0.853, and 0.494-0.774 for the Bluetooth, Browser, Calendar, Camera and MMS datasets respectively. These ranges of AUC values indicate that the categorization of defects into different levels according to the SDC models developed using the maintenance effort required to correct them is acceptable and accurate. An analysis of the AUC values of 'low level', 'medium level' and 'high level' SDC

models indicate that the 'high level' category models obtained higher AUC and accuracy values on three of the datasets (Browser, Calendar, and Camera). As mentioned earlier, it is important to detect 'high level' defects according to the maintenance effort so that maintenance team managers can allocate appropriate resources for correcting these defects. This will help in reducing the cost to fix defects and so effective maintenance of the software with optimized resource usage.

The AUC values for 'low level' SDC models ranged from 0.753-0.913, 0.585-0.653, 0.569-0.837, 0.610-0.826 and 0.421-0.834, while the AUC values of 'medium level' SDC models were found to be in the range of 0.750-0.891, 0.574-0.669, 0.482-0.788, 0.626-0.815 and 0.448-0.726 in most of the cases in Bluetooth, Browser, Calendar, Camera and MMS datasets respectively. The AUC values for 'high level' SDC models were found to be in the range of 0.687-0.881, 0.540-0.604, 0.621-0.900, 0.713- 0.853, and 0.494-0.774 respectively in Bluetooth, Browser, Calendar, Camera and MMS datasets respectively. These AUC values indicate that the categorization of defects into different levels according to the corresponding maintenance effort required to correct them is acceptable and accurate.

B. SDC Model using Change Impact

The AUC values of SDC models developed in accordance with Change Impact using the NBM classification technique on the android data sets used in the study are shown in Table 4.3. All AUC values with a value greater than 0.7 are maked as bold in Table 4.3. The AUC values for 'low level' SDC models ranged from 0.513-0.680, 0.554-0.643, 0.588-0.801, 0.629-0.759 and 0.535-0.644, while the AUC values of 'medium level' SDC models were found to be in the range of 0.489-

0.607, 0.476-0.517, 0.487-0.591, 0.514-0.627 and 0.546-0.64 in most of the cases in Bluetooth, Browser, Calendar, Camera and MMS datasets respectively. The AUC values for 'high level' SDC models were found to be in the range of 0.418-0.665, 0.536-0.662, 0.660-0.812, 0.652-0.765 and 0.367-0.678 respectively in Bluetooth, Browser, Calendar, Camera and MMS datasets.

As seen in the Table 4.3, most of the AUC values are accurate and acceptable, but, few SDC models obtained AUC values in the range 0.3-0.5. These poor values were obtained as the SDC models were developed from highly imbalanced training data.

C. SDC Model according to the combined effect of its Maintenance Effort and Change Impact

SDC models were developed using the NBM classification technique in accordance with the product of a bug's required maintenance effort and its change impact values. Table 4.4 states the AUC results of the developed SDC models. All AUC values greater than 0.7 are marked in bold.

As shown in the Table 4.4, the SDC models built at 'high' level category obtained AUC values in the range 0.612-0.807, 0.545-0.688, 0.654-0.889, 0.672-0.846 and 0.655-0.870 in most of the cases for Bluetooth, Browser, Calendar, Camera and MMS datasets respectively. The SDC models at 'low level' obtained the ranges of AUC values of from 0.739-0.908, 0.575-0.705, 0.643-0.848, 0.638-0.794 and 0.670-0.871, while those at 'medium level' obtained the ranges of AUC values of 0.664-0.796, 0.570-0.658, 0.571-0.815, 0.523-0.766 and 0.676-0.861 respectively for Bluetooth, Browser, Calendar, Camera and MMS datasets. Most of the obtained AUC values by the SDC models depicted in Table 4.4 were acceptable.

Table 4.1: AUC values of SDC Models based on Maintenance Effort using Multinomial Naïve bayes

Defect Level	Bluetooth				Browser				Calendar				Camera				MMS			
	Top10	Top25	Top50	Top100	Top10	Top25	Top50	Top5	Top10	Top25	Top50	Top10	Top25	Top50	Top100	Top10	Top25	Top50	Top100	
Low	**0.753**	**0.799**	**0.913**	**0.874**	0.585	0.597	0.653	0.569	0.681	**0.818**	**0.837**	0.61	**0.702**	**0.759**	**0.826**	0.421	**0.72**	**0.77**	**0.834**	
Medium	**0.75**	**0.833**	**0.868**	**0.891**	0.574	0.604	0.669	0.482	0.563	**0.741**	**0.788**	0.626	0.694	**0.724**	**0.815**	0.448	0.6	0.641	**0.726**	
High	0.687	**0.833**	**0.825**	**0.881**	0.573	0.54	0.604	0.621	0.695	**0.871**	**0.9**	**0.713**	**0.795**	**0.805**	**0.853**	0.494	0.643	0.673	**0.774**	

Table 4.2: AUC values of SDC Models based on Change Impact using Multinomial Naïve bayes

Defect Level	Bluetooth				Browser				Calendar				Camera				MMS			
	Top10	Top25	Top50	Top100	Top10	Top25	Top50	Top100	Top10	Top25	Top50	Top100	Top10	Top25	Top50	Top100	Top10	Top25	Top50	Top100
Low	**0.577**	**0.68**	**0.609**	**0.513**	0.567	0.554	0.566	0.643	0.588	**0.724**	**0.774**	**0.801**	**0.629**	**0.672**	**0.716**	**0.759**	**0.535**	**0.541**	**0.644**	0.632
Medium	**0.489**	**0.592**	**0.607**	**0.581**	0.511	0.476	0.502	0.517	0.487	**0.591**	**0.557**	0.562	0.514	**0.592**	**0.586**	0.627	0.615	0.625	**0.642**	0.546
High	0.418	**0.518**	**0.67**	**0.665**	0.557	0.536	0.552	0.662	0.66	**0.812**	**0.783**	**0.812**	**0.652**	**0.743**	**0.756**	**0.765**	0.367	0.552	**0.593**	0.678

Table 4.3: AUC values of SDC Models based on Maintenance Effort and Change Impact using Multinomial Naïve bayes

Defect Level	Bluetooth				Browser				Calendar				Camera				MMS			
	Top10	Top25	Top50	Top100	Top10	Top25	Top50	Top100	Top10	Top25	Top50	Top100	Top10	Top25	Top50	Top100	Top10	Top25	Top50	Top100
Low	**0.739**	**0.807**	**0.908**	**0.9**	0.575	0.586	0.625	**0.705**	0.643	**0.81**	**0.801**	**0.848**	**0.638**	**0.67**	**0.755**	**0.794**	**0.67**	**0.735**	**0.819**	**0.871**
Medium	**0.664**	**0.796**	**0.714**	**0.731**	0.578	0.57	0.624	0.658	0.571	**0.713**	**0.747**	**0.815**	0.523	**0.635**	**0.707**	**0.766**	0.676	**0.782**	**0.836**	**0.861**
High	0.612	**0.807**	**0.698**	**0.683**	0.545	0.551	0.574	0.688	0.654	**0.819**	**0.882**	**0.889**	**0.672**	**0.784**	**0.799**	**0.846**	0.655	**0.753**	**0.768**	**0.87**

4.3.2 Discussion of Results

The results of the study show that the AUC values of the SDC models built using the combination of maintenance effort and change impact was comparable to SDC models built in accordance with the maintenance effort. As the combined approach considers both the maintenance effort and the change impact values, they found to be more useful. Maintenance effort aids in planning the developer's effort based on the level of defects predicted, while the change impact aid in planning the testerâs effort based on the prediction of the number of the classes that are changed to fix the defects. Testers can perform regression testing on only the modules that are impacted, which saves cost and time. The results of the study also show that the AUC values of the SDC models built using the combined approach were better than SDC models built in accordance with change impact values. The primary reason for the poor performance of SDC models based on change impact values was the imbalanced data of the training dataset.

It can also be observed from the results of the study that all the performance of the developed SDC models for Top10 features was poor as compared to Top25, Top50, and Top100 features. In most of the instances, the AUC of SDC models developed using Top10 keywords were lesser than Top25, Top50 or Top100 words. The primary reason for such poor performance was the inability of Top10 words to encapsulate the required information for predicting the level of a defect based on its bug report. On the other hand, both the SDC models developed using Top50 and Top100 keywords were found appropriate with effective AUC values. The reason for the comparable performance of Top50 and Top100 models could be redundancy in the predictors. There is a possibility that Top100 words may be representing redundant information in certain cases, which

was effectively encapsulated in just Top50 words. In such cases, developing models with Top50 words is more.

The results of this study are as follows:

- The results of AUC values of 'high' category SDC models on all the datasets used in the study for Top 25, Top 50 and Top 100 words as predictors are summarized as follows: The most of AUC values of SDC models built in accordance of maintenance effort are in the range 0.643-0.881 for all the datasets of the study. The most of AUC values of SDC models built based on change impact values were in the range 0.652-0.812. The most of AUC values of SDC models built based on the product of maintenance effort and change impact are in the range of 0.654-0.889. These AUC values indicate acceptable predictability of the SDC models built in our study. Similar values were shown in Table 4.3 and Table 4.4 for 'medium' category and 'low' category SDC models respectively.

- The performance of the SDC models according to the combination of maintenance effort and change impact are found to be better than the SDC models based on change impact and comparable to the SDC models according to maintenance effort.

4.4 SDC with Ensemble Learners

Machine learning, especially ensemble learning techniques, has emerged as a promising avenue for classification problems [143][18]. These ensemble techniques have the capability to handle the complexities of real-world bug datasets and provide robust defect categorization models. Ensemble learning is a machine learning paradigm that combines

the predictions of multiple base learners to produce a more accurate prediction. The basic idea behind ensemble learning is that by combining the predictions of multiple base learners, the variance of the overall prediction can be reduced [144]. This is because different base learners will make different errors, and combining their predictions will average out these errors. There are two main types of ensemble learning techniques: bagging and boosting. Bagging techniques work by training multiple base learners on different subsets of the training data. Boosting techniques work by training multiple base learners sequentially, with each base learner learning from the errors of the previous base learner. The mathematical foundation of ensemble learning is rooted in concepts of aggregation, weighting, and diversity. The brief description of ensemble learning techniques examined in this study are given in Chapter 2.

The primary aim of this study is to assess the effectiveness of ensemble learning techniques in the context of software defect categorization, with a specific focus on enhancing categorization accuracy and efficiency. The study aims to understand how these techniques can be harnessed to address the inherent challenges of categorizing software bugs in real-world scenarios.

The following research questions are addressed in this study:

RQ1. What is the performance of ensemble learning techniques for software defect categorization.?

RQ2. How does the performance of ensemble learning techniques vary for different levels of bugs.?

RQ3. How does the performance of ensemble learning techniques vary for different bug categorization approaches.?

RQ4. Which ensemble learning technique outperforms the other ensemble techniques for software defect categorization.?

The ensemble learning techniques employed for building SBC models in this study are Random Forest (RF), eXtreme Gradient Boosting (XGBoost), Adaptive Boosting (AdaBoost), and Bootstrap Aggregation or Bagging (BAG). Thus, a total of 5 (defect datasets) x 4 (sets of independent variables) x 3 (approaches) x 4 (techniques) = 240 SDC models were built during experimentation.

4.4.1 Experimental Framework

This section detail the experimental setup employed to assess the effectiveness of ensemble learning techniques for software defect categorization (SDC). The implementation of ensemble learning models for SBC heavily relied on the scikit-learn (Sci-kit) library [61], a powerful and widely used machine learning library in Python. Scikit-learn provides a versatile set of tools for constructing and evaluating machine learning models, making it an ideal choice for this research. The tables in this section, namely Table 4.5 for Random Forest (RF), Table 4.6 for eXtreme Gradient Boosting (XGBoost), Table 4.7 for Adaptive Boosting (AdaBoost), and Table 4.8 for Bagging (BAG), display the parameter configurations that were utilized during the construction of the classifiers. It's worth noting that default parameter values provided by the scikit-learn library were employed for any parameters not explicitly mentioned in these tables. Additionally, IBM SPSS [145] was utilized for performing the statistical tests - Friedman test and Wilcoxon signed-rank test.

Table 4.4: Parameter Configuration of Random Forest Classifier

Parameter	Description	Value
criterion	The function to measure the quality of a split. Supported criteria are 'gini' for the Gini impurity and 'log_loss' and 'entropy' both for the Shannon information gain.	gini
max_depth	The maximum depth of the tree	5
max_features	The number of features to consider when looking for the best split:	sqrt
min_impurity_decrease	A node will be split if this split induces a decrease of the impurity greater than or equal to this value.	0
min_samples_leaf	The minimum number of samples required to be at a leaf node.	1
min_samples_split	The minimum number of samples required to split an internal node.	2
min_weight_fraction_leaf	The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node.	0
n_estimators	The number of trees in the forest.	100

Table 4.5: Parameter Configuration of XGBoost Classifier

Parameter	Description	Value
objective	Specify the learning task and the corresponding learning objective or a custom objective function to be used.	binary:logistic
n_estimators	Number of boosting rounds.	100

Table 4.6: Parameter Configuration of AdaBoost Classifier

Parameter	Description	Value
algorithm	If 'SAMME.R' then use the SAMME.R real boosting algorithm. If 'SAMME' then use the SAMME discrete boosting algorithm.	SAMME.R
estimator	The base estimator from which the boosted ensemble is built.	DecisionTreeClassifier
learning_rate	Weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier.	1
n_estimators	The maximum number of estimators at which boosting is terminated.	100

Table 4.7: Parameter Configuration of Bagging Classifier

Parameter	Description	Value
bootstrap	Whether samples are drawn with replacement. If False, sampling without replacement is performed.	TRUE
estimator	The base estimator to fit on random subsets of the dataset.	DecisionTreeClassifier
max_features	The number of features to draw from X to train each base estimator.	1
max_samples	The number of samples to draw from X to train each base estimator.	1
n_estimators	The number of base estimators in the ensemble.	100

4.4.2 Results

This section provides an overview of the study's findings and addresses each of the research questions (RQ) under investigation. The primary goal of Software Defect Categorization (SDC) models is to effectively categorize software defects into one of three specific levels: ”low”, ”medium,” or ”high.” To achieve this, four distinct models with each ensemble classification algorithms, namely Random Forest (RF), eXtreme Gradient Boosting (XGB), Adaptive Boosting (ADB), and Bagging (BAG), were developed for every bug level and every set of predictor variables (Top10, Top25, Top50, and Top100). These models are built upon five distinct datasets drawn from the study. Table 4.9, Table 4.10, and Table 4.11 presents the AUC values associated with the SBC models based on maintenance effort (ME), change impact (CI) and combination of both (COMB) respectively. The AUC value of the most effective model for each predictor variable set is highlighted in bold. Furthermore, we computed the average AUC values across the four ensemble algorithms, denoted as AVG, for each predictor variable set (Top10, Top25, Top50, and Top100). The best AVG AUC value among these sets is also highlighted in bold.

Table 4.8: AUC values of SBC Models based on Maintenance Effort using Ensemble Learners

Dataset	Features	Low Level Defects					Medium Level Defects					High Level Defects				
		RF	**XGB**	**ADB**	**BAG**	***AVG***	**RF**	**XGB**	**ADB**	**BAG**	***AVG***	**RF**	**XGB**	**ADB**	**BAG**	***AVG***
Bluetooth	Top10	0.7150	0.6908	**0.7767**	0.7383	*0.7302*	0.6083	0.6067	**0.7000**	0.6600	*0.6438*	0.5683	0.4950	**0.6300**	0.5383	*0.5579*
	Top25	0.7242	0.6992	**0.7575**	0.7225	*0.7259*	0.7142	0.6525	0.7683	**0.7742**	*0.7273*	0.6667	0.6250	**0.6883**	0.6367	*0.6542*
	Top50	**0.8050**	0.7567	0.7833	0.7250	***0.7675***	0.6500	0.6458	**0.7433**	0.6967	*0.6840*	0.6742	0.5892	**0.6758**	0.6325	*0.6429*
	Top100	0.6800	0.6475	0.5667	**0.7408**	*0.6588*	**0.5850**	0.4708	0.5692	0.5733	*0.5496*	0.5900	0.6050	0.5850	**0.6567**	*0.6092*
Browser	Top10	0.5866	**0.5947**	0.5672	0.5756	*0.5810*	**0.5803**	0.5280	0.4797	0.5555	*0.5359*	0.6540	0.6449	0.6521	**0.6548**	*0.6515*
	Top25	**0.6452**	0.6270	0.6033	0.6075	*0.6208*	**0.5839**	0.5799	0.5468	0.5777	*0.5721*	0.6809	0.6571	**0.6909**	0.6713	*0.6751*
	Top50	**0.6514**	0.6407	0.6177	0.6274	*0.6343*	0.5932	0.5696	0.5885	**0.5967**	*0.5870*	0.6996	0.6799	**0.7034**	0.6894	*0.6931*
	Top100	0.6501	0.6527	0.5913	**0.6670**	*0.6403*	0.5628	**0.5891**	0.5642	0.5823	*0.5746*	0.6901	**0.7131**	0.6637	0.6904	*0.6893*
Calendar	Top10	0.4826	0.4668	**0.5841**	0.4826	*0.5040*	0.6011	0.6178	**0.6253**	0.5995	*0.6109*	0.6368	0.6217	0.6253	**0.6495**	*0.6333*
	Top25	0.5605	0.5559	**0.6468**	0.5859	*0.5873*	0.5961	**0.6078**	0.5961	0.5978	*0.5995*	0.6703	0.6136	0.6612	**0.6833**	*0.6571*
	Top50	0.7233	0.6848	**0.7248**	0.7148	*0.7119*	0.6625	0.6203	**0.6700**	0.6433	*0.6490*	0.7236	0.6965	**0.7347**	0.6882	*0.7108*
	Top100	0.7261	0.6173	**0.7600**	0.6805	*0.6960*	**0.6139**	0.4486	0.5843	0.4766	*0.5309*	**0.8148**	0.6436	0.7674	0.6039	*0.7074*
Camera	Top10	0.6213	0.6056	**0.6460**	0.6061	*0.6198*	**0.6413**	0.6098	0.6238	0.5953	*0.6176*	**0.7189**	0.7118	0.7042	0.7147	*0.7124*
	Top25	0.6562	0.6442	**0.6624**	0.6414	*0.6511*	**0.6082**	0.6017	0.5788	0.5785	*0.5918*	**0.7601**	0.7590	0.7357	0.7479	*0.7507*
	Top50	0.7032	0.6979	0.6376	**0.7132**	*0.6880*	**0.6322**	0.6139	0.6260	0.6120	*0.6210*	**0.7521**	0.7365	0.7038	0.6958	*0.7221*
	Top100	0.6898	**0.7013**	0.6659	0.6965	*0.6884*	0.6421	0.6177	0.6407	**0.6562**	*0.6392*	**0.7697**	0.7468	0.7211	0.7110	*0.7372*
MMS	Top10	0.6702	0.7011	0.6939	**0.7139**	*0.6948*	0.5995	**0.6030**	0.5208	0.5907	*0.5785*	0.6722	**0.6937**	0.6552	0.6536	*0.6687*
	Top25	0.7681	0.7285	0.7305	**0.7831**	*0.7526*	0.5411	0.5377	0.5077	**0.5795**	*0.5415*	0.6333	0.6185	**0.6436**	0.6377	*0.6333*
	Top50	0.7168	0.7008	0.6639	**0.7286**	*0.7025*	0.5813	**0.6272**	0.5242	0.6105	*0.5858*	0.6493	0.6368	**0.5542**	0.6386	*0.6197*
	Top100	**0.7299**	0.6862	0.6473	0.6961	*0.6899*	0.5963	0.6044	0.5819	**0.6365**	*0.6048*	**0.6751**	0.6420	0.6239	0.6156	*0.6392*

Table 4.9: AUC values of SBC Models based on Change Impact using Ensemble Learners

Dataset	Features	Low Level Defects					Medium Level Defects					High Level Defects				
		RF	**XGB**	**ADB**	**BAG**	***AVG***	**RF**	**XGB**	**ADB**	**BAG**	***AVG***	**RF**	**XGB**	**ADB**	**BAG**	***AVG***
Bluetooth	Top10	0.4775	0.5358	**0.5392**	0.4675	*0.5050*	0.4399	**0.5036**	**0.4149**	0.3768	*0.4338*	**0.6655**	0.6083	**0.5798**	0.6512	*0.6262*
	Top25	0.5658	0.5408	**0.4542**	**0.5950**	*0.5390*	0.6060	0.5565	**0.6667**	**0.6083**	*0.6094*	0.6595	0.6583	**0.6607**	0.5821	*0.6402*
	Top50	**0.5508**	0.4783	0.5733	0.4925	***0.5237***	**0.6667**	0.6274	**0.5875**	0.6054	*0.6218*	**0.5786**	0.4702	**0.4298**	0.4524	*0.4828*
	Top100	0.5667	0.5617	**0.6233**	**0.5633**	*0.5788*	**0.5321**	0.5131	0.5149	0.4976	*0.5144*	**0.6643**	0.6571	0.6500	**0.6429**	*0.6536*
Browser	Top10	**0.5598**	**0.5514**	0.5611	0.5574	*0.5574*	**0.4217**	0.4381	**0.4755**	0.4312	*0.4416*	0.5936	0.5864	**0.6072**	**0.5985**	*0.5964*
	Top25	**0.6012**	0.5661	0.5033	0.5495	*0.5550*	**0.4480**	0.4856	0.4553	**0.4955**	*0.4711*	**0.5940**	0.5485	**0.5912**	0.5633	*0.5743*
	Top50	**0.6364**	0.5845	0.5097	0.5894	*0.5800*	0.4689	0.4550	**0.4820**	**0.4545**	*0.4651*	**0.6228**	0.6033	**0.5770**	0.5762	*0.5948*
	Top100	**0.6205**	0.6054	0.5487	**0.6020**	*0.5942*	0.4236	**0.4199**	**0.4931**	0.4582	*0.4487*	**0.6337**	**0.6275**	0.6187	0.6122	*0.6230*
Calendar	Top10	**0.5975**	0.5708	**0.5561**	0.5863	*0.5777*	0.5740	**0.5955**	**0.5638**	0.5849	*0.5796*	**0.6507**	0.6241	0.5861	**0.6211**	*0.6205*
	Top25	**0.7010**	0.6064	**0.6454**	0.6817	*0.6586*	0.6548	**0.5929**	0.6705	**0.6753**	*0.6484*	**0.7592**	0.6818	0.7141	**0.7345**	*0.7224*
	Top50	0.7015	0.6706	**0.7246**	0.6729	*0.6924*	0.6442	0.6054	**0.6699**	0.6279	*0.6369*	**0.7789**	0.7335	**0.6978**	0.7493	*0.7399*
	Top100	0.7213	0.6785	**0.6229**	**0.7393**	*0.6905*	**0.6035**	0.6429	**0.6619**	0.6231	*0.6329*	**0.7485**	0.7483	0.7066	0.7411	*0.7361*
Camera	Top10	**0.6389**	0.6123	**0.5810**	0.6172	*0.6124*	**0.5503**	**0.5639**	0.5171	0.5285	*0.5400*	**0.6686**	0.6496	0.6533	0.6531	*0.6562*
	Top25	**0.6881**	0.6779	**0.5954**	0.6632	*0.6562*	**0.5714**	**0.5962**	0.5163	0.5667	*0.5627*	**0.7250**	**0.7495**	0.7198	0.7119	*0.7266*
	Top50	0.6979	**0.7059**	0.5830	**0.7050**	*0.6730*	**0.5995**	**0.6179**	0.4807	0.5555	*0.5634*	**0.7405**	0.7356	0.6907	0.6981	*0.7162*
	Top100	**0.6832**	**0.6634**	0.6040	0.6787	*0.6573*	0.5767	**0.6077**	0.5686	**0.5437**	*0.5742*	**0.7220**	0.7038	0.6940	0.6857	*0.7014*
MMS	Top10	**0.7125**	0.6755	0.6001	**0.6988**	*0.6717*	0.5344	**0.5406**	**0.5903**	0.5112	*0.5441*	**0.7097**	**0.6811**	0.6579	0.6480	*0.6742*
	Top25	**0.7018**	0.6463	0.5558	**0.6874**	*0.6478*	0.6155	0.6132	**0.6495**	**0.5946**	*0.6182*	**0.8127**	0.7769	**0.6483**	0.7742	*0.7530*
	Top50	**0.7055**	0.6938	0.5611	**0.6810**	*0.6604*	**0.6649**	**0.6264**	0.6435	0.6187	*0.6384*	**0.8098**	0.7874	**0.5222**	0.8032	*0.7307*
	Top100	**0.7027**	0.6814	0.5278	0.6633	*0.6438*	**0.7329**	0.5759	0.5955	**0.6639**	*0.6421*	**0.8010**	0.7565	0.5994	0.7774	*0.7336*

Table 4.10: AUC values of SDC Models based on Combined Approach using Ensemble Learners

Dataset	Features	Low Level Defects					Medium Level Defects					High Level Defects				
		RF	**XGB**	**ADB**	**BAG**	***AVG***	**RF**	**XGB**	**ADB**	**BAG**	***AVG***	**RF**	**XGB**	**ADB**	**BAG**	***AVG***
Bluetooth	Top10	**0.7592**	0.665	**0.7325**	0.7392	*0.724*	0.6708	**0.6625**	**0.7325**	0.6842	*0.6875*	**0.52**	0.445	**0.5633**	0.5333	*0.5154*
	Top25	0.6983	0.6483	**0.72**	**0.73**	*0.6992*	0.75	0.665	**0.7667**	**0.7583**	*0.735*	0.6092	0.5242	**0.6775**	0.6358	*0.6117*
	Top50	**0.7792**	0.6925	**0.7792**	0.7425	***0.7484***	**0.5417**	0.5267	**0.64**	0.595	*0.5759*	**0.5992**	0.4892	**0.6292**	0.5525	*0.5675*
	Top100	0.6842	0.6325	**0.5525**	**0.7058**	*0.6438*	**0.545**	0.5217	**0.6117**	0.5467	*0.5563*	**0.52**	0.53	0.5283	**0.6033**	*0.5454*
Browser	Top10	**0.5965**	**0.5907**	0.589	0.591	*0.5918*	**0.55**	0.5254	**0.5047**	**0.5414**	*0.5304*	**0.6359**	0.6053	**0.6072**	**0.6153**	*0.6159*
	Top25	**0.6511**	0.6275	0.6128	0.6022	*0.6234*	**0.544**	0.5657	0.5799	**0.5884**	*0.5695*	**0.6665**	0.6041	**0.6571**	0.6425	*0.6426*
	Top50	**0.6563**	0.6416	0.6193	0.6252	*0.6356*	**0.5842**	0.564	**0.5771**	**0.5835**	*0.5772*	**0.6956**	0.6524	**0.6943**	0.656	*0.6746*
	Top100	**0.6616**	0.6539	0.6252	**0.6655**	*0.6516*	**0.592**	**0.5704**	**0.5579**	0.5646	*0.5712*	**0.6845**	**0.6673**	0.6761	0.6748	*0.6757*
Calendar	Top10	**0.6211**	0.615	**0.6214**	**0.6271**	*0.6212*	0.5217	**0.5508**	**0.5792**	0.5733	*0.5563*	**0.6202**	0.557	0.6024	**0.6238**	*0.6009*
	Top25	**0.7141**	0.702	**0.6505**	**0.727**	*0.6984*	0.5942	**0.545**	**0.6592**	**0.6092**	*0.6019*	**0.7379**	0.6467	0.7062	**0.6858**	*0.6942*
	Top50	0.6718	0.6533	**0.6823**	0.6186	*0.6565*	0.5583	0.5142	**0.5967**	0.535	*0.5511*	**0.7052**	0.6353	**0.6918**	0.658	*0.6726*
	Top100	**0.735**	0.705	**0.7253**	**0.7211**	*0.7216*	**0.595**	0.5475	**0.5933**	0.5892	*0.5813*	**0.7173**	0.6523	0.6135	0.6374	*0.6551*
Camera	Top10	**0.6558**	0.6464	**0.6487**	0.651	*0.6505*	**0.5519**	**0.5164**	0.5345	0.507	*0.5275*	**0.6948**	0.6778	0.6885	0.6815	*0.6857*
	Top25	**0.6707**	0.6386	**0.6238**	0.6353	*0.6421*	**0.5548**	**0.5392**	0.5078	0.5516	*0.5384*	**0.7426**	**0.7651**	0.7333	0.7416	*0.7457*
	Top50	0.6918	**0.6824**	0.6232	**0.7024**	*0.675*	**0.6104**	**0.5813**	0.5804	0.5858	*0.5895*	**0.7337**	**0.7483**	0.6947	0.7158	*0.7231*
	Top100	**0.6632**	**0.6989**	0.6292	**0.7153**	*0.6767*	0.6037	**0.6138**	0.5957	**0.6147**	*0.607*	**0.7307**	**0.7345**	0.699	0.7022	*0.7166*
MMS	Top10	**0.7189**	**0.7213**	0.7083	**0.7181**	*0.7167*	**0.6764**	**0.6724**	**0.6258**	0.6585	*0.6583*	**0.7036**	**0.7295**	0.7066	0.6898	*0.7074*
	Top25	**0.7613**	0.722	0.7375	**0.7722**	*0.7483*	0.7233	0.7211	**0.713**	**0.742**	*0.7249*	**0.7692**	0.6731	**0.7108**	0.7499	*0.7258*
	Top50	**0.6879**	**0.7102**	0.6483	**0.6941**	*0.6851*	**0.6762**	**0.7015**	0.6931	**0.7302**	*0.7003*	**0.7612**	0.6899	**0.6512**	0.726	*0.7071*
	Top100	**0.6882**	**0.703**	0.603	0.7097	*0.676*	**0.6746**	0.6971	0.7042	**0.7453**	*0.7053*	**0.7544**	0.7098	0.6265	0.709	*0.6999*

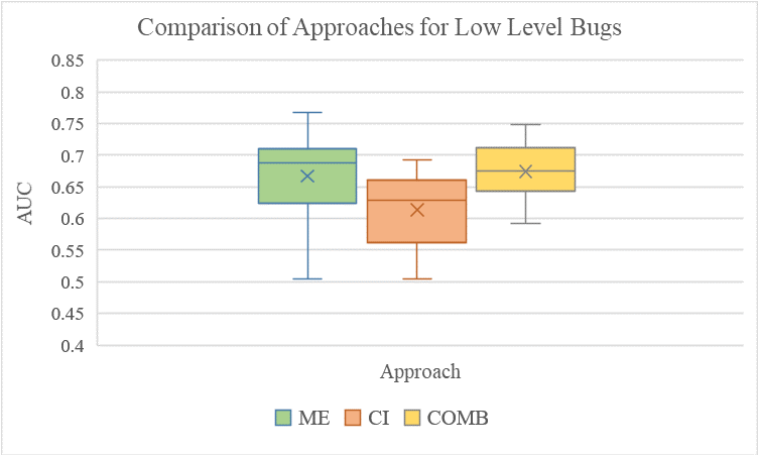

Figure 4.2: Box plot of AUC values of different Approaches for Low Level Bugs

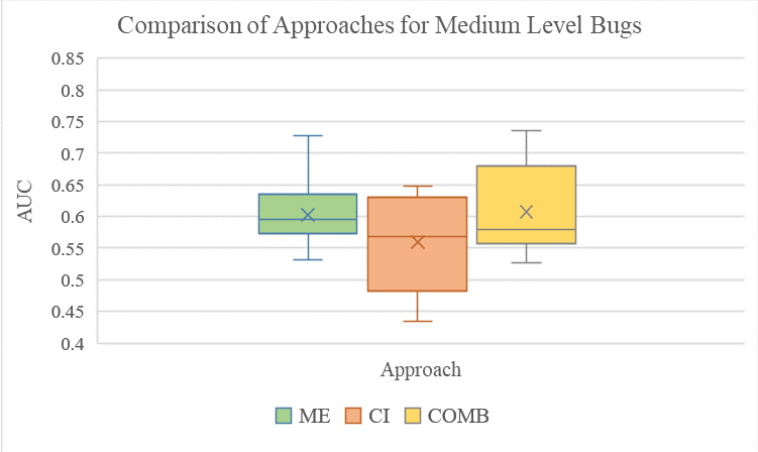

Figure 4.3: Box plot of AUC values of different Approaches for Medium Level Bugs

- RQ1: What is the performance of ensemble learning techniques for software defect categorization.?

  It is evident from the AUC values presented in Table 4.9, Table 4.10, and Table

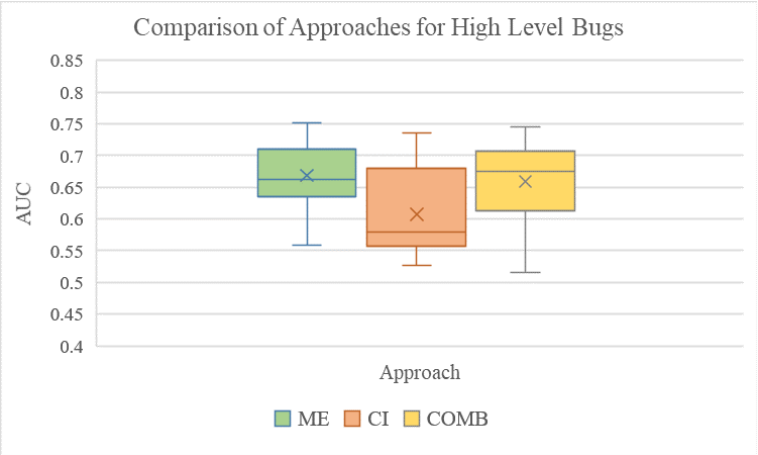

Figure 4.4: Box plot of AUC values of different Approaches for High Level Bugs

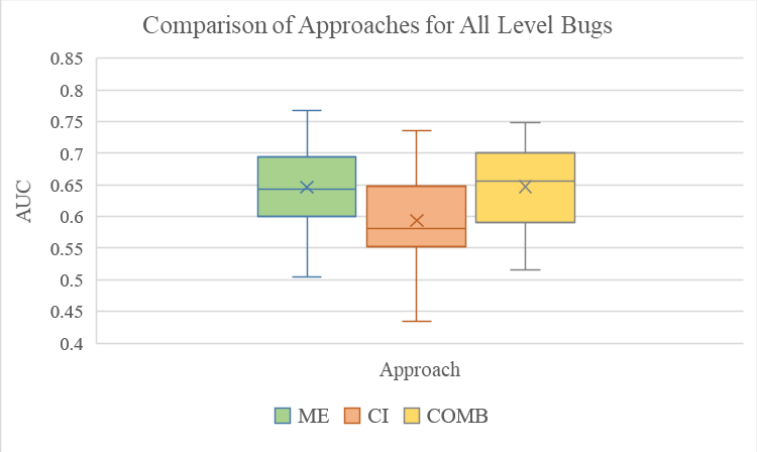

Figure 4.5: Box plot of AUC values of different Approaches for All Level Bugs

4.11 that the range of AUC values for RF, XGB, ADB and BAG are 0.4217-0.8148, 0.4199-0.7874, 0.4149-0.7833, and 0.3768-0.8032 respectively. The average AUC value for RF, XGB, ADB and BAG are 0.6499, 0.6263, 0.6262 and 0.6382 respectively. AUC values exceeding 0.5 were observed in 96.11% of cases for RF,

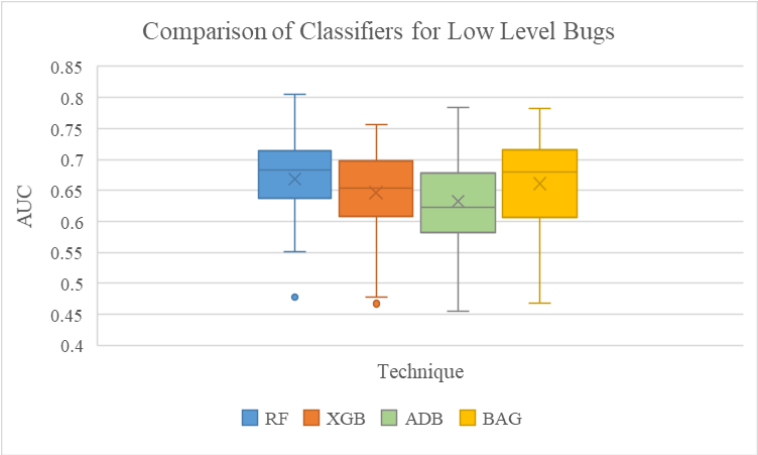

Figure 4.6: Box plot of AUC values of Ensemble Techniques for Low Level Bugs

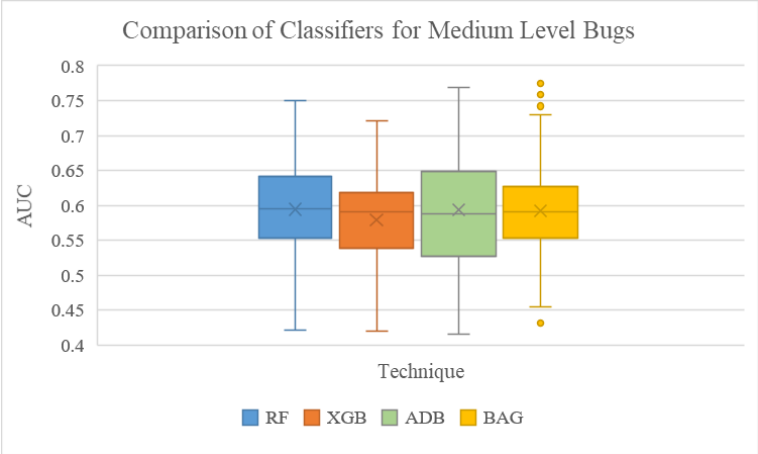

Figure 4.7: Box plot of AUC values of Ensemble Techniques for Medium Level Bugs

93.33% for XGB, 95% for ADB, and 93.89% for BAG. These AUC values signify the effectiveness and predictive capability of the ensemble classifiers for the bug categorization.

- RQ2: How does the performance of ensemble learning techniques vary for different

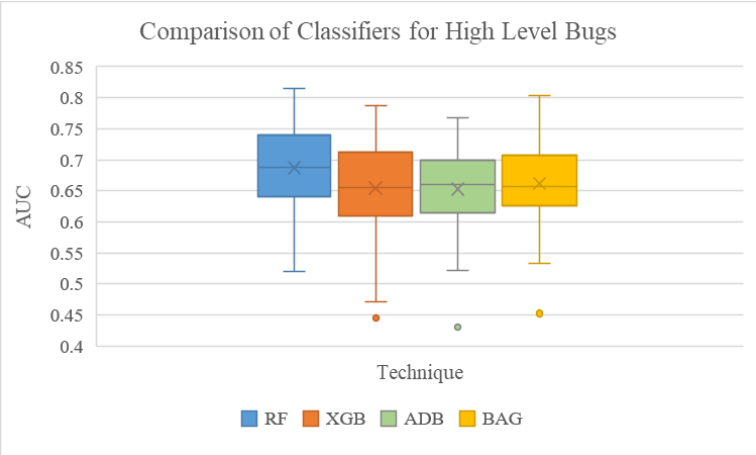

Figure 4.8: Box plot of AUC values of Ensemble Techniques for High Level Bugs

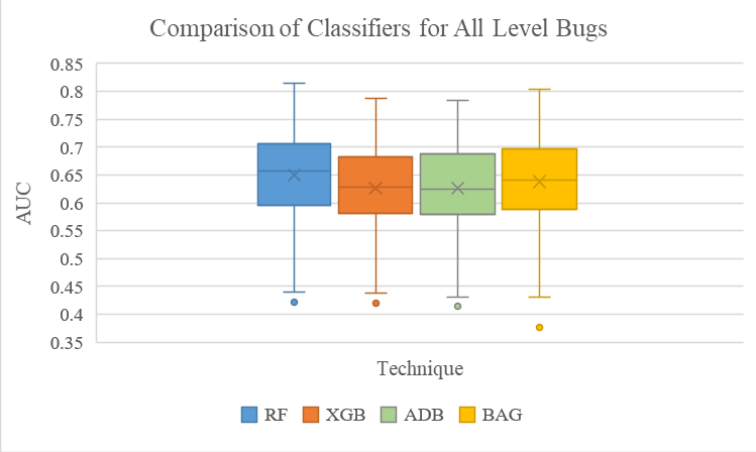

Figure 4.9: Box plot of AUC values of Ensemble Techniques for All Level Bugs

levels of bugs.?

The AUC values obtained for different levels of bugs (”low level”, ”medium level” and ”high level”) are shown in Table 4.9, Table 4.10, and Table 4.11. The box plots of AUC values of ensemble learning techniques over ”low level”, ”medium level”,

”high level” and ”all” bugs are illustrated in Figure 4.6, Figure 4.7, Figure 4.8 and Figure 4.9 respectively. The AUC value ranges for the ”low level” bugs of SDC models built on RF, XGB, ADB and BAG are 0.4775-0.8050, 0.4668-0.7567, 0.4542-0.7833 and 0.4675-0.7831 respectively, and their respective average AUC values are 0.6684, 0.6459, 0.6321 and 0.6605. Meanwhile, the AUC values of ”medium level” bugs ranges of 0.4217-0.7500 (RF), 0.4199-0.7211 (XGB), 0.4149-0.7683 (ADB) and 0.3768-0.7742 (BAG), and their respective average AUC values are 0.5940, 0.5789, 0.5935 and 0.5920. The ”high level” bugs of SBC models demonstrate AUC value ranges of 0.5200-0.8148, 0.4450-0.7874, 0.4298-0.7674 and 0.4524-0.8032 for the RF, XGB, ADB and BAG respectively, and their respective average AUC values are 0.6873, 0.6542, 0.6530 and 0.6620. These AUC values signify the effectiveness and predictive capability of the ensemble classifiers for the categorization of different levels of bugs. It is worth highlighting that the models categorize ”high level” bugs exhibit higher AUC values for all the ensemble learners. The effective identification and resolution of ”high level” bugs by considering their maintenance effort are essential for the efficient allocation of resources and cost reduction in bug resolution, ultimately contributing to the optimization of software maintenance.

- RQ3: How does the performance of ensemble learning techniques vary for different bug categorization approaches.?

  Table 4.9 illustrates the AUC values obtained from the Software Defect Categorization (SDC) models based on the Maintenance Effort (ME) by applying four ensemble classification algorithms. The AUC value ranges for the ”low level” SDC models across the Bluetooth, Browser, Calendar, Camera, and MMS datasets are

0.5667-0.8050, 0.5672-0.6670, 0.4668-0.7600, 0.6056-0.7132 and 0.6473-0.7831 respectively. Meanwhile, the ”medium level” SDC models exhibit AUC value ranges of 0.4708-0.7742 (Bluetooth), 0.4797-0.5967 (Browser), 0.4486-0.6700 (Calendar), 0.5785-0.6562 (Camera) and 0.5077-0.6365 (MMS). The ”high level” SDC models demonstrate AUC value ranges of 0.4950-0.6883, 0.6449-0.7131, 0.6039-0.8148, 0.6958-0.7697 and 0.5542-0.6937 for the Bluetooth, Browser, Calendar, Camera, and MMS datasets respectively. Likewise, Table 4.10 and Table 4.11 present the AUC ranges achieved by the SBC models when categorized by Change Impact (CI) and the combined approach (COMB) respectively. The box plots of AUC values of three approaches (ME, CI and COMB) over ”low level”, ”medium level”, ”high level” and ”all” bugs are illustrated in Figure 4.2, Figure 4.3, Figure 4.4 and Figure 4.5 respectively.

To assess the comparative performance of the three distinct approaches - Maintenance Effort (ME), Change Impact (CI), and their combination (COMB), the study conducted a comprehensive statistical analysis using the Friedman test, which evaluated their AUC values across all datasets examined in this study. The significance level chosen for this analysis was set at $\alpha = 0.05$.

Null Hypothesis H_{01}: There is no statistically significant difference among the performance of the defect categorization approaches - ME, CI, and COMB in terms of AUC.

Alternative Hypothesis H_{a1}: There exists a statistically significant difference among the performance of the defect categorization approaches - ME, CI, and COMB in terms of AUC.

Table 4.12 summarizes the results of the Friedman test, assessing the performance of bug categorization approaches - ME, CI, and COMB, as measured by AUC. The obtained p-value was less than 0.05, signifying statistically significant outcomes. Consequently, we reject the null hypothesis H_{01}, which posits that the performance of the bug categorization approaches - ME, CI, and COMB is identical. Table 4.12 also provides the mean ranks assigned to each of the bug categorization approaches - ME, CI, and COMB. It is noteworthy that ME achieved the highest rank, COMB secured the next rank and CI attained the lowest rank across all bug levels.

To further validate the findings of the Friedman test, the study conducted a post-hoc analysis using the Wilcoxon signed-rank test. This analysis compared the performance of ME with that of the other approaches (COMB and CI). The null and alternative hypotheses for the Wilcoxon signed-rank test concerning AUC in this study were formulated as follows:

Null Hypothesis H_{02}: Performance of X = Performance of Y

Alternative Hypothesis H_{a2}: Performance of X $\neq$ Performance of Y where X represents ME and COMB, and Y represents COMB and CI.

The study applied a significance level of α = 0.05 and implemented Bonferroni correction to assess and potentially reject null hypotheses H_{02}. The study compared two pairs of techniques using the Wilcoxon test, whereby null hypotheses would be rejected if the obtained p-value exceeded 0.05. Table 4.13 displays the outcomes of the Wilcoxon signed-rank test, along with the corresponding test statistics. The ”S+” column indicates a significant difference in the AUC values for a pair of compared methods, while ”S-” suggests no significant difference for the respective pair of

methods. The results of the Wilcoxon signed-rank test reveal the following:

- There is no significant difference between the ME and COMB approaches.
- Both ME and COMB significantly outperform CI.

- RQ4: Which ensemble learning technique outperforms the other ensemble techniques for software defect categorization.?

  The performance of ensemble models (Random Forest (RF), eXtreme Gradient Boost (XGA), AdaBoost (ADB) and Bagging (BAG)) are assessed using Area under ROC curve (AUC) given in Table 4.9, Table 4.10 and Table 4.11. The comparison of ensemble learning techniques over ”low level”, ”medium level”, ”high level” and ”all” bugs are illustrated as box plots in Figure 4.6, Figure 4.7, Figure 4.8 and Figure 4.9 respectively. The results clearly indicate the RF has better AUC values, followed by BAG, then ADB and XGB lastly, for most of the datasets. Statistical Tests are performed to further strengthen the results. To investigate which of these four ensemble techniques viz., RF, XGA, ADB and BAG performs the best, we conducted the statistical analysis using the Friedman test by evaluating their AUC values over all datasets examined in this study. The Friedman test is applied at a level of significance α = 0.05.

  Null hypothesis-H_{03}: There is no significant difference among the performance of SBC models by ensemble techniques viz., RF, XGA, ADB and BAG in terms of AUC.

  Alternate hypothesis-H_{a3}: There is a significant difference among the performance of SBC models by ensemble techniques viz., RF, XGA, ADB and BAG in terms of

AUC.

Table 4.14 presents the outcomes of Friedman test conducted to evaluate the performance of SBC models by classification techniques - RF, XGA, ADB and BAG measured by AUC. The obtained p-value was less than 0.05, which prove the outcomes are significant. Therefore, we reject the null hypothesis H_{03}, which states the performance of SBC models by classification techniques - RF, XGA, ADB and BAG are same. The mean ranks assigned to each of the prediction models classification techniques - RF, XGA, ADB and BAG are presented in Table 4.14. It is to be noted that RF has obtained the best rank, followed by BAG, then ADB and XGB at the last.

To confirm the finding of the Friedman test, we carried out post-hoc analysis through the Wilcoxon-signed rank test. The analysis compared the performance of one technique with that of other techniques. The null and alternative hypothesis for the Wilcoxon-signed rank test in terms of AUC in this study are presented as follows:

Null hypothesis-H_{04}: Performance of X = Performance of Y

Alternate hypothesis-H_{a4}: Performance of X $\neq$ Performance of Y where X and Y denotes RF, XGA, ADB and BAG.

The study used a level of confidence α = 0.05 and Bonferroni correction to reject null hypotheses H_{03}. The study compared 2 pairs of techniques by Wilcoxon test, the null hypotheses would be rejected if p-value obtained is greater than 0.05. Table 4.15 presents outcomes of Wilcoxon signed-rank test along with test statistics. The "S+" column in Table 4.15 shows significant difference in the AUC values of a pair of compared techniques, while "S-" denotes that there is no significant difference in

the corresponding pair of methods. The outcomes of the Wilcoxon signed-rank test indicate that:

- ***RF technique is significantly superior to other techniques (BAG, ADB and XGA).***
- ***BAG is significantly superior to ADB and XGB.***
- ***ADB and XGB are not significantly different.***

Table 4.11: Results of Friedman Test - Comparison of Approaches

Approach	Mean Rank over			
	High Level Bugs	Medium Level Bugs	Low Level Bugs	All Bugs
ME	2.45	2.30	2.40	2.38
COMB	2.05	2.10	2.35	2.17
CI	1.50	1.60	1.25	1.45

Table 4.12: Results of Wilcoxon Signed Rank Test - Comparison of Approaches

Approaches Comparison	Test Statistics over			
	High Level Bugs	Medium Level Bugs	Low Level Bugs	All Bugs
ME vs COMB	S- (0.313)	S- (0.940)	S- (0.881)	S- (0.375)
ME vs CI	S+ (0.021)	S+ (0.037)	S+ (0.005)	S+ (0.000)
COMB vs CI	S+ (0.028)	S+ (0.019)	S+ (0.000)	S+ (0.000)

Table 4.13: Results of Friedman Test - Comparison of Classifiers

Classifier	Mean Rank over			
	High Level Bugs	Medium Level Bugs	Low Level Bugs	All Bugs
RF	3.43	2.74	3.17	3.11
BAG	2.22	2.55	2.76	2.51
ADB	2.23	2.51	1.99	2.24
XGB	2.12	2.20	2.08	2.13

Table 4.14: Results of Wilcoxon Signed Rank Test - Comparison of Classifiers

Classifiers	Test Statistics over			
Comparison	High Level Bugs	Medium Level Bugs	Low Level Bugs	All Bugs
RF vs XGB	S+ (0.000)	S+ (0.018)	S+ (0.000)	S+ (0.000)
RF vs ADB	S+ (0.000)	S- (0.982)	S+ (0.000)	S+ (0.000)
RF vs BAG	S+ (0.000)	S- (0.740)	S+ (0.038)	S+ (0.000)
XGB vs ADB	S- (0.897)	S- (0.108)	S- (0.093)	S- (0.845)
XGB vs BAG	S- (0.219)	S+ (0.011)	S+ (0.002)	S+ (0.000)
ADB vs BAG	S- (0.549)	S- (0.791)	S+ (0.003)	S+ (0.032)

4.4.3 Discussion

The study explored the application of ensemble learning techniques of Software Defect Categorization (SDC) models on the datasets of five modules of the android operating system. The SDC models are capable to estimate whether a software bug belongs to ”low category” or ”not low category”, ”medium category” or ”not medium category”, ”high category” or ”not high category”. This study embarked on the categorization of bugs on basis of three distinct approaches: maintenance effort, change impact, and their combination, with the intent to unveil the most potent approach for constructing resilient SBC models. The study explored and compared four prominent ensemble learning methods, namely Random Forest (RF), eXtreme Gradient Boosting (XGB), Adaptive Boosting (ADB) and Bagging (BAG), to determine the best ensemble technique for effective SDC model development. Thus, a total of 5 (datasets) x 4 (sets of independent variables) x 3 (approaches) x 4 (ensemble methods) = 240 SBC models were built and evaluated in this study. The study harnessed the rigorous power of stratified ten-fold cross-validation and performance measure Area Under the Curve (AUC) to unveil the true essence of these

models, and their predictive capabilities.

The key findings of the study are summarized as below:

- ***The AUC values of three approaches - maintenance effort, change impact, and their combination are in the range of 0.5040-0.7675, 0.4338-0.7350 and 0.5154-0.7484 respectively.***
- ***The performance of SDC models based on combined approach and maintenance effort are superior to models based on change impact. Also, the performance of SDC models based on maintenance effort are comparable to the models based on combined approach.***
- ***The AUC values of four ensemble learners - Random Forest, XGBoost, AdaBoost and Bagging are in the range of 0.4217-0.8148, 0.4199-0.7874, 0.4149-0.7833 and 0.3768-0.8032 respectively.***
- ***Among the four ensemble learning techniques examined in the study, Random Forest exhibits superior performance, followed by Bagging in the second position, Ada-Boost in the third position, and XGBoost in the fourth and final position.***
- ***The results signify the effectiveness and predictive capability of the ensemble classifiers for the software defect categorization.***

4.5 Convolutional Neural Networks for Software Defect Categorization

Deep learning methods have surged in popularity, driven by advancements in computing capabilities and storage capacity. One prominent deep learning architecture that has gained prominence is the Convolutional Neural Network (CNN), which is inspired by the structure and functioning of the visual cortex in the human brain. This study introduces a novel software defect categorization model built upon a Convolutional Neural Network (SDC-CNN). The study is driven by two primary objectives:

- To determine the predictive capability of convolutional networks for software defect categorization. This objective entails a comprehensive exploration and comparative analysis of performance of models for each defect level - low, medium and high.
- To determine the most effective attribute for constructing robust Software Defect Categorization (SDC) models. This involves systematically evaluating and comparing three key attributes: maintenance effort, change impact, and a combination of both attributes to identify the optimal attribute for defect categorization.

To achieve these objectives, the study poses several research questions:

RQ1: What is the predictive capability of SDC-CNN models for high level defects.?

RQ2: What is the predictive capability of SDC-CNN models for medium level defects.?

RQ3: What is the predictive capability of SDC-CNN models for low level defects.?

RQ4: What is the performance of SDC-CNN models that assign defect levels based on maintenance effort.?

RQ5: What is the performance of SDC-CNN models that assign defect levels based on change impact.?

RQ6: What is the performance of SDC-CNN models that assign defect levels based on the combined effect of maintenance effort and change impact.?

RQ7: What is the comparative performance of SDC-CNN models using the combined effect of maintenance effort and change impact, compared to levels allocated based on a) maintenance effort and b) change impact.?

The study provides a comprehensive and reproducible framework for efficiently implementing CNN, offering a detailed account of parameter configurations and employing a stratified 10-fold cross-validation method. In total, the study involves the development of 5 (datasets) x 4 (feature sets) x 3 (approaches) = 60 SDC models.

4.5.1 Proposed Convolutional Neural Network for Software Defect Categorization Model (SDC-CNN)

Table 4.16 outlines the CNN's architecture along with its parameter configurations. The CNN implementation is carried out using python APIs - keras [146] and sci-kit [61]. The architecture and model parameters were chosen based on a heuristic approach. The models begin with the input layer, which is configured to expect data in the form of 1D arrays, denoted as (X, 1). Moving forward, a 1D convolutional layer with 64 filters and a kernel

size of 4 is employed to capture essential features from the input defect data. The smaller kernel size focuses on localized patterns, which can be crucial for identifying specific defect characteristics. ReLU (Rectified Linear Unit) is used as the activation function in the convolutional and dense layers to introduce non-linearity, enabling the network to learn complex mappings between defect features and their categories. The output from this convolutional layer is then directed into a max-pooling layer with a pool size of 2. This step serves two purposes: reducing the complexity of the feature maps and guarding against overfitting. Another significant aspect of max-pooling is its ability to introduce a degree of translation invariance. This means that even if a feature is detected in different spatial locations within the input (e.g., different positions in a defect report), max-pooling will capture the most prominent occurrence of that feature. This property is beneficial in defect categorization tasks where defects may manifest in varying contexts or locations within the input data. Then the flatten layer is used to transform the output of convolutional and pooling layers into a suitable format for the next dense layer featuring 128 units and employing ReLU activation.

To further mitigate the risk of overfitting, a dropout layer is introduced with a dropout rate of 0.25, randomly deactivating 25% of input units during training. Then Batch Normalization layers are used to stabilize and accelerate the training of CNNs by normalizing the input data and introducing learnable parameters to fine-tune the normalization. Following this, the output of the aforementioned layer proceeds to another dense layer, this time comprising a three units (each unit corresponds to a different defect level - low, medium and high) and equipped with a softmax activation function. This configuration generates a probability score indicating the likelihood of the input belonging to one of three classes.

Table 4.15: Structure of proposed CNN

Layer	Layer Type	Output Shape	Parameter Settings
0	Input	(None, length)	Batch Size : 50
1	Convolutional 1D	(None,length,64)	Filter Size : 64, Kernel Size : 4, Regularization : L2 with penalty of 0.001, Activation Function : ReLU
2	Pooling	(None,length,64)	1D Max Pooling with pool size 2
3	Flatten	(None, 192)	data_format : channels_last
4	Dense	(None, 128)	Regularization : L2 with penalty of 0.001, Activation Function : ReLU
5	Dropout	(None, 128)	Dropout 0.25
6	Batch Normalization	(None, 128)	axis : -1, momentum : 0.99, epsilon : 0.001
7	Dense	(None, 3)	Regularization : L2 with penalty of 0.001, Activation Function : Softmax
Loss Function : Categorical Cross entropy Optimizer : Adam			

4.5.2 Results

This section presents the results of the study by providing answers to each of the research questions (RQ) that have been investigated.

- *RQ1: What is the predictive capability of SDC-CNN models for high level defects.?*

  The AUC values of SDC-CNN models for high level defects based on maintenance effort, change effort and combined approach are presented in Table 4.17, Table 4.18 and Table 4.19 respectively. Boxplot of AUC values of the models for high level defects are shown in Figure 4.10. These models constructed for each level using varying predictor variables (Top10, Top25, Top50, and Top100). Values greater than 0.7 are highlighted in bold. For the 'high-level' SDC models across

Bluetooth, Browser, Calendar, Camera, and MMS datasets, AUC value ranges based on maintenance effort, change effort and combined approaches were 0.5150-0.8577, 0.4548-0.8031, and 0.5550-0.8586 respectively.

Notably, the 'high-level' category models achieved AUC values greater than 0.5 in most of the cases. The predictive capability of the proposed SDC-CNN model for high level defects is found to be very significant. The predictive capability of SDC-CNN models for 'high-level' defects underscores their importance in identifying critical issues that can significantly impact software functionality and performance. This implies that organizations can use SDC-CNN models to prioritize the resolution of high-level defects through efficient resource allocation by maintenance managers, thereby reducing defect fixing costs and optimizing software maintenance.

- *RQ2: What is the predictive capability of SDC-CNN models for medium level defects.?*

  The AUC values of SDC-CNN models for medium level defects based on maintenance effort, change effort and combined approach are presented in Table 4.17, Table 4.18 and Table 4.19 respectively. Boxplot of AUC values of the models for medium level defects are shown in Figure 4.11. These models constructed for each level using varying predictor variables (Top10, Top25, Top50, and Top100). Values greater than 0.7 are highlighted in bold. For the 'medium-level' SDC models across Bluetooth, Browser, Calendar, Camera, and MMS datasets, AUC value ranges based on maintenance effort, change effort and combined approaches were 0.5084-0.7906, 0.4395-0.6974, and 0.5357-0.8967 respectively.

  Notably, the 'medium-level' category models achieved values of AUC greater than

0.5 in most of the cases. The predictive capability of the proposed SDC-CNN model for medium level defects is found to be significant. Understanding the predictive capability of SDC-CNN models for 'medium-level' defects highlights their effectiveness in addressing issues that may not be as critical as 'high-level' defects but still require attention. This suggests that organizations can leverage SDC-CNN models to allocate resources more efficiently and prioritize defect resolution efforts based on the severity of defects.

- *RQ3: What is the predictive capability of SDC-CNN models for low level defects.?*

  The AUC values of SDC-CNN models for low level defects based on maintenance effort, change effort and combined approach are presented in Table 4.17, Table 4.18 and Table 4.19 respectively. Boxplot of AUC values of the models for low level defects are shown in Figure 4.12. These models constructed for each level using varying predictor variables (Top10, Top25, Top50, and Top100). Values greater than 0.7 are highlighted in bold. For the 'low-level' SDC models across Bluetooth, Browser, Calendar, Camera, and MMS datasets, AUC value ranges based on maintenance effort, change effort and combined approaches were 0.5763-0.8280, 0.4983-0.7723, and 0.5580-0.8067 respectively.

  Notably, the 'low-level' category models achieved values of AUC greater than 0.5 in most of the cases. The predictive capability of the proposed SDC-CNN model for low level defects is found to be significant. The predictive capability of SDC-CNN models for low-level defects indicates their value in identifying minor issues that may have a cumulative effect on software performance and user experience over time. This implies that organizations can use SDC-CNN models to proactively address

low-level defects, thereby preventing potential escalations and reducing long-term maintenance costs.

- *RQ4: What is the performance of SDC-CNN models that assign defect levels based on maintenance effort.?*

  The AUC values of SDC-CNN models that assign defect levels based on maintenance effort are presented in Table 4.17. These models classify defects into 'low,' 'medium,' or 'high' categories, with four models constructed for each level using varying predictor variables (Top10, Top25, Top50, and Top100). Values of AUC greater than 0.7 are highlighted in bold. The AUC value ranges of SDC models across Bluetooth, Browser, Calendar, Camera, and MMS datasets were 0.5150-0.8058, 0.5084-0.6869, 0.6000-0.8577, 0.5358-0.8107 and 0.6713-0.8075 respectively. The AUC values of all the cases are greater than 0.5 with an average AUC value of 0.6981.

  Evaluating the performance of SDC-CNN models that assign defect levels based on maintenance effort suggests their potential to optimize resource allocation and improve the efficiency of defect management processes. This implies that organizations can use SDC-CNN models to prioritize defect resolution activities based on the effort required, thereby streamlining maintenance workflows and maximizing productivity.

- *RQ5: What is the performance of SDC-CNN models that assign defect levels based on change impact.?*

  The AUC of SDC-CNN models that assign defect levels based on change impact are presented in Table 4.18. These models classify defects into 'low,' 'medium,' or

'high' categories, with four models constructed for each level using varying predictor variables (Top10, Top25, Top50, and Top100). Values of AUC greater than 0.7 are highlighted in bold. The AUC value ranges of SDC models across Bluetooth, Browser, Calendar, Camera, and MMS datasets were 0.4548-0.6726, 0.4395-0.6524, 0.5298-0.8031, 0.5497-0.7643 and 0.4937-0.7964 respectively. The AUC values of 55 out of 60 cases are greater than 0.5 with an average AUC value of 0.6245. Thus the performance of SDC-CNN models that assign defect levels based on change impact is found to be significant.

Assessing the performance of SDC-CNN models that assign defect levels based on change impact highlights their effectiveness in identifying defects that have a significant impact on software functionality. This suggests that organizations can use SDC-CNN models to proactively address high-impact defects, thereby enhancing software stability and minimizing disruptions.

- *RQ6: What is the performance of SDC-CNN models that assign defect levels based on the combined effect of maintenance effort and change impact.?*

  The AUC values of SDC-CNN models that assign defect levels based on combination of maintenance effort and change impact are presented in Table 4.19. These models classify defects into 'low,' 'medium,' or 'high' categories, with four models constructed for each level using varying predictor variables (Top10, Top25, Top50, and Top100). Values of AUC greater than 0.7 are highlighted in bold. The AUC value ranges of SDC models across Bluetooth, Browser, Calendar, Camera, and MMS datasets were 0.5550-0.8967, 0.5357-0.7028, 0.5580-0.8586, 0.6017-0.7852 and 0.5598-0.8039 respectively. All the AUC values are greater than 0.5 with an

average AUC value of 0.7008. All the AUC values are greater than 0.4 with an average AUC value of 0.5633. Thus the performance of SDC-CNN models that assign defect levels based on maintenance effort is found to be very significant.

Investigating the performance of SDC-CNN models that assign defect levels based on the combined effect of maintenance effort and change impact underscores the synergistic benefits of integrating multiple factors in defect categorization. This implies that organizations can use SDC-CNN models to enhance the accuracy and effectiveness of defect prediction models by leveraging both maintenance effort and change impact metrics.

- *RQ7: What is the comparative performance of SDC-CNN models using the combined effect of maintenance effort and change impact, compared to levels allocated based on a) maintenance effort and b) change impact?*

  The AUC values of maintenance effort (ME), change impact (CI) and combination of maintenance effort and change impact (COMB) are given in Table 4.17, Table 4.18 and Table 4.19 respectively. The results clearly indicate the COMB has better AUC values to ME and CI for most of the datasets. Statistical Tests are performed to further strengthen the results. To investigate which of these three approaches viz., ME, CI, and COMB performs the best, we conducted the statistical analysis using the Friedman test by evaluating their performance over all datasets examined in this study. The Friedman test is applied at a level of significance $\alpha = 0.05$.

  Null hypothesis-H_{01}: There is no significant difference among the performance of defect categorization approaches - ME, CI, and COMB.

  Alternate hypothesis-H_{a1}: There is a significant difference among the performance

of defect categorization approaches - ME, CI, and COMB.

First, the performance of each approach (ME, CI, COMB) is assessed based on their values of performance measures over all datasets. The approaches are ranked individually across all datasets based on their performance. In this study, higher ranks indicate better performance, so an approach with consistently higher ranks across datasets is considered superior. For each approach, we calculated the average rank across all datasets. This is done by summing up the ranks of an approach across datasets and dividing by the total number of datasets. Once mean ranks are calculated for each approach, we compared these mean ranks. The approach with the highest mean rank is considered the best-performing approach across the datasets. Higher mean ranks indicate more consistent and superior performance compared to lower mean ranks. The results of the Friedman test are presented in Table 4.20, where the obtained p-value was less than 0.05, indicating the significance of the outcomes. Therefore, we rejected the null hypothesis H_{01}, which suggests that the performance of the defect categorization approaches - ME, CI, and COMB - is not the same. Further insight into their performance is provided by the mean ranks assigned to each approach, with Combined approach having the best rank, followed by maintenance effort, and change impact receiving the lowest rank.

To corroborate the Friedman test's finding that COMB was the superior defect categorization approach, we conducted a post-hoc analysis using the Wilcoxon-signed rank test. This analysis compared COMB's performance with that of the other approaches (ME and CI). The null and alternative hypotheses for the Wilcoxon-signed rank test in this study were as follows:

Null hypothesis-H_{02}: Performance of COMB = Performance of X

Alternate hypothesis-H_{a2}: Performance of COMB $\neq$ Performance of X

(where X denotes ME and CI)

A significance level of α = 0.05 and Bonferroni correction were applied to reject null hypotheses H_{02} in 2 out 3 cases. Comparing two pairs of techniques via the Wilcoxon test, we would reject the null hypothesis if the p-value obtained is greater than 0.05. Table 4.21 presents the results of the Wilcoxon signed-rank test, including test statistics. The 'S+' column indicates a significant difference in the values of performance measures between a pair of compared methods, while 'S-' denotes no significant difference between the respective pair of methods. The outcomes of the Wilcoxon signed-rank test reveal that:

- The combined approach and maintenance effort are significantly superior change impact.
- Although combined approach is having edge over maintenance effort, there is no significant difference between them.

Comparing the performance of SDC-CNN models using the combined effect of maintenance effort and change impact with levels allocated based solely on maintenance effort or change impact provides insights into the relative effectiveness of different categorization approaches. This suggests that organizations can use a combined approach of considering both maintenance effort and change impact metrics to improve the accuracy and reliability of defect prediction and management

Table 4.16: AUC values of SDC-CNN Models based on Maintenance Effort

Module	Feature Set	Defect Level		
		Low	Medium	High
Bluetooth	Top10	**0.7258**	0.6700	0.5150
	Top25	**0.7892**	**0.7567**	**0.7467**
	Top50	**0.8058**	**0.7817**	**0.7233**
	Top100	**0.8008**	**0.7500**	0.5750
Browser	Top10	0.5763	0.5084	0.5862
	Top25	0.6427	0.5883	0.6386
	Top50	0.6655	0.5986	0.6529
	Top100	0.6869	0.6258	0.6634
Calendar	Top10	0.6515	0.6000	0.6500
	Top25	**0.7532**	0.6833	**0.7902**
	Top50	**0.7494**	0.6433	**0.8145**
	Top100	**0.8280**	**0.7817**	**0.8577**
Camera	Top10	0.6269	0.5358	0.6738
	Top25	0.6658	0.5686	**0.7234**
	Top50	**0.7071**	0.5931	**0.7572**
	Top100	**0.7768**	**0.7193**	**0.8107**
MMS	Top10	**0.7027**	0.6744	0.6713
	Top25	**0.7617**	**0.7124**	**0.7261**
	Top50	**0.7617**	**0.7209**	**0.7294**
	Top100	**0.7917**	**0.7906**	**0.8075**

Table 4.17: AUC values of SDC-CNN Models based on Change Impact

Module	Feature Set	Defect Level		
		Low	Medium	High
	Top10	0.5142	0.6089	0.4548

Bluetooth

Table 4.17 continued from previous page

Module	Feature Set	Defect Level		
		Low	Medium	High
	Top25	0.5133	0.5369	0.5988
	Top50	0.5983	0.6726	0.6488
	Top100	0.4983	0.5762	0.5619
Browser	Top10	0.5382	0.4395	0.5752
	Top25	0.5155	0.4509	0.5714
	Top50	0.5302	0.494	0.5589
	Top100	0.6066	0.4826	0.6524
Calendar	Top10	0.588	0.5298	0.63
	Top25	0.671	0.6974	**0.7213**
	Top50	0.6814	0.6298	**0.7359**
	Top100	**0.7723**	0.6827	**0.8031**
Camera	Top10	0.6267	0.5497	0.661
	Top25	0.6682	0.5954	**0.7397**
	Top50	0.6817	0.5962	**0.7643**
	Top100	**0.7187**	0.603	**0.7264**
MMS	Top10	0.6863	0.6047	**0.745**
	Top25	0.6673	0.6434	**0.7934**
	Top50	**0.7235**	0.6038	**0.784**
	Top100	0.6586	0.4937	**0.7964**

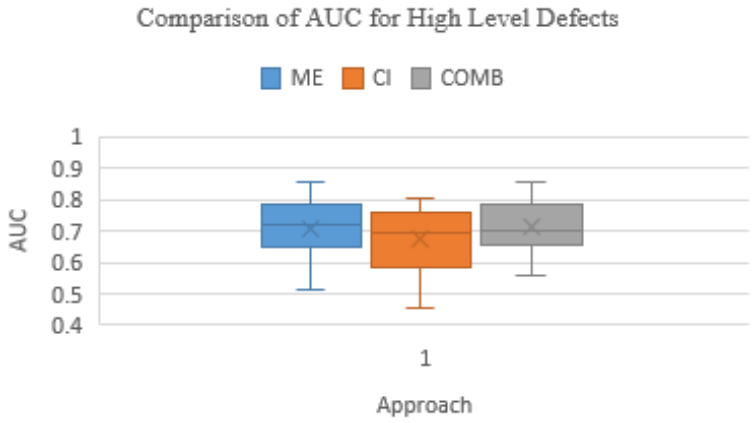

Figure 4.10: Box plot of AUC values of SDC-CNN for High Level Defects

Table 4.18: AUC values of SDC-CNN Models based on Combined Approach

Module	Feature Set	Defect Level		
		Low	Medium	High
Bluetooth	Top10	**0.7175**	**0.735**	0.555
	Top25	**0.7142**	**0.8058**	**0.7875**
	Top50	**0.8067**	**0.8383**	**0.7908**
	Top100	**0.795**	**0.8967**	**0.8317**
Browser	Top10	0.5779	0.5357	0.6133
	Top25	0.6208	0.5479	0.6724
	Top50	0.6658	0.6259	0.685
	Top100	**0.7028**	0.68	0.6983
Calendar	Top10	0.558	0.6386	0.6614
	Top25	0.6241	0.6714	**0.7106**
	Top50	**0.7944**	**0.7156**	**0.8173**
	Top100	**0.8038**	0.695	**0.8586**
Camera	Top10	0.6017	0.6109	**0.7077**
	Top25	0.673	0.6456	**0.7551**
	Top50	**0.7236**	0.6909	**0.7643**
	Top100	**0.7654**	**0.763**	**0.7852**
MMS	Top10	0.6961	0.6139	0.6655
	Top25	**0.7779**	0.5598	0.6208
	Top50	**0.8039**	0.6436	0.6279
	Top100	**0.7902**	0.6594	0.6532

Table 4.19: Results of Friedman Test

Approach	Mean Rank for High Defects	Mean Rank for Medium Defects	Mean Rank for Low Defects	Mean Rank Overall
COMB	2.425	2.55	2.3	2.43
ME	2.05	2.15	2.5	2.23
CI	1.525	1.3	1.2	1.34

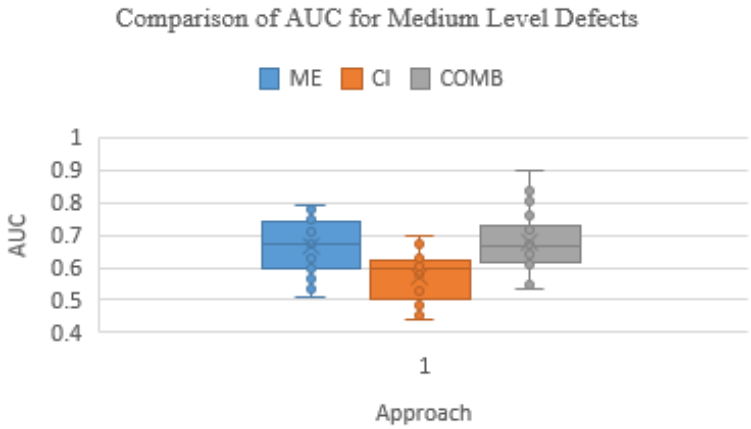

Figure 4.11: Box plot of AUC values of SDC-CNN for Medium Level Defects

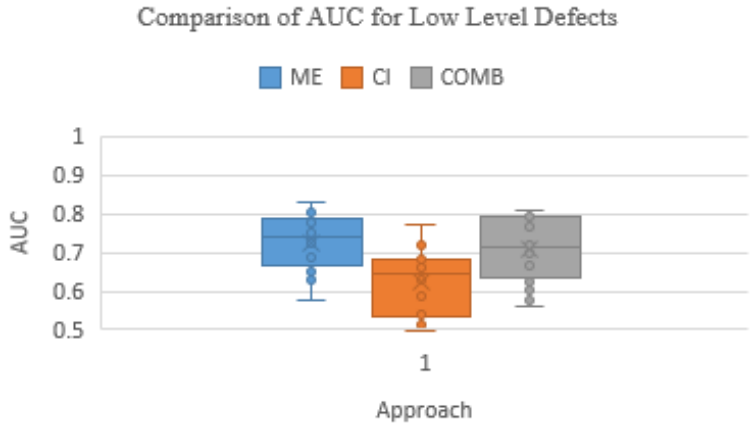

Figure 4.12: Box plot of AUC values of SDC-CNN for Low Level Defects

Table 4.20: Results of Wilcoxon Signed Rank Test

Approach	Test Statistics for High Defects	Test Statistics for Medium Defects	Test Statistics for Low Defects	Test Statistics Overall
COMB vs ME	S- (0.331)	S- (0.455)	S- (0.351)	S- (0.351)
COMB vs CI	S+ (0.036)	S+ (0.000)	S+ (0.001)	S+(0.036)
ME vs CI	S+ (0.026)	S+ (0.001)	S+ (0.000)	S+ (0.026)

4.5.3 Discussion

The study developed Software Defect Categorization (SDC) models based on three defect attributes: maintenance effort, change impact, and their combined effect. These models classify software defects into 'low,' 'medium,' or 'high' categories by mining keywords from defect reports. SDC models, employing Convolutional Neural Networks (CNN), were created for each scenario, using feature sets (Top10, Top25, Top50, and Top100) for five Android Operating System Modules. Thus, a total of 5 (datasets) x 4 (feature sets) x 3 (approaches) = 60 SDC models were developed in this study. The results, validated using values of the three performance metrics - Area Under the Curve (AUC), Accuracy and MCC values, are summarized as follows:

- For 'high' category SDC models, AUC values ranged from 0.5150-0.8577 for maintenance effort, 0.4548-0.8031 for change impact, and 0.5550-0.8586 for the combined approach, indicating good predictability. Further, results are supported by the values of Accuracy - 0.5084-0.7906, 0.4395-0.6974, and 0.5357-0.896, and MCC - 0.5763-0.8280, 0.4983-0.7723, and 0.5580-0.8067 for maintenance effort, change impact and combine approach respectively.

- For 'medium' category SDC models, AUC values ranged from 0.5084-0.7906 for maintenance effort, 0.4395-0.6974 for change impact, and 0.5357-0.8967 for the combined approach, indicating good predictability. Further, results are supported by the values of Accuracy - 0.4992-0.7003, 0.4590-0.6324, and 0.5244-0.7794, and MCC - 0.5632-0.7265, 0.4881-0.6862, and 0.5230-0.7068 for maintenance effort, change impact and combine approach respectively.

- For 'low' category SDC models, AUC values ranged from 0.5763-0.8280 for maintenance effort, 0.4983-0.7723 for change impact, and 0.5580-0.8067 for the combined approach, indicating good predictability. Further, results are supported by the values of Accuracy - 0.3761-0.6489, 0.3216-0.5548, and 0.4082-0.7541 and MCC - 0.4568-0.6865, 0.3662-0.6311, and 0.4131-0.6584 for maintenance effort, change impact and combine approach respectively.

The combined approach, considering both maintenance effort and change impact, showed superior performance compared to models based solely on change impact and was on par with maintenance effort-based models.These SDC models can aid software practitioners in estimating developer and tester efforts, optimizing resource allocation, and managing costs. For instance, 'high' category defects, identified by maintenance effort, but 'low' in change impact, require more developer effort and less testing resources, while the reverse applies to defects categorized as 'low' in maintenance effort and 'high' in change impact. Therefore, the performance of the Software Defect Categorization (SDC) models utilizing the CNN algorithm appears promising.Researchers are encouraged to validate these models on different datasets, domains, and platforms, explore alternative classification algorithms, and assess generalizability.

Chapter 5

Techniques for Hyperparameter Optimization in Software Quality Models: A Systematic Review

5.1 Introduction

Software systems have experienced unprecedented growth in size and complexity, presenting significant challenges in terms of building high quality software while minimizing costs [31]. To address these challenges, the early prediction of defect-prone components, maintenance requirements, effort estimation, and reliability has become crucial. Predictive modeling, a technique that generates models to forecast future outcomes, has gained popularity in software engineering as a means to address these needs [147]. By analyzing historical and current data, predictive modeling extracts predictive rules that can be applied

to future data, enabling effective planning and resource allocation. Various classification/learning algorithms, including statistical methods, machine learning, and evolutionary algorithms, are utilized for developing these models [120][8]. The performance of predictive models depends on several factors, and researchers have been actively working to improve their effectiveness. One crucial factor influencing model performance is the choice of hyperparameter values in the learning algorithms used to build the prediction models[87]. Hyperparameters influence the behavior and performance of the learning algorithm. For example, the hyperparameters of a neural network include the learning rate (α), the number of hidden units (h), and the number of epochs (E). In the context of this paper, the term ”hyperparameter” will be used interchangeably with ”parameter,” and the process of tuning hyperparamters will also be referred to as ”parameter optimization.”

It has been observed that different parameter values used during the construction of classifiers can result in significant variance in performance[87]. To address this issue, researchers have explored parameter tuning techniques to identify the optimal settings within the parameter space of classification algorithms. By finetuning the hyperparameters, researchers aim to minimize variance and improve the performance of classifiers. Parameter tuning involves systematically exploring different combinations of hyperparameter values to find the most suitable configuration. Several studies have demonstrated that models developed using optimal parameter settings exhibit improved performance compared to those built with default parameter values. The problem of hyperparameter tuning has received significant attention from researchers, leading to the proposal of various approaches aimed at ad-dressing this challenge. One of the widely adopted approaches is Grid Search, which systematically explores a predefined set of hyperparameter values and evaluates the performance of the model for each combination [148]. It provides a brute-force method to

exhaustively search the hyperparameter space and identify the best configuration based on a chosen evaluation metric. Another popular technique is Random Search, which randomly samples hyperparameter values from predefined ranges [149]. Unlike Grid Search, which systematically explores all possible combinations of hyperparameters within specified ranges, Random Search takes a more probabilistic approach. It strategically selects hyperparameter values at random from different regions within the predefined ranges. In a Random Search, each hyperparameter configuration is chosen independently, and the search space is randomly sampled over multiple iterations. This targeted randomness enables a more efficient exploration of the hyperparameter space by prioritizing promising areas without the need to evaluate every potential combination. The advantage lies in the ability to discover optimal or near-optimal hyperparameter configurations more quickly, making Random Search a favorable choice in scenarios where computational resources are limited. Bayesian Optimization is another noteworthy approach that combines prior knowledge with observed performance to guide the search for optimal hyperparameters[150]. It utilizes a probabilistic model to model the unknown performance function and suggests promising regions for further exploration. Evolutionary Algorithms, such as Genetic Algorithms, mimic the process of natural selection to iteratively search for optimal hyperparameter configurations [151][148]. These algorithms maintain a population of candidate solutions and use genetic operators, such as crossover and mutation, to generate new configurations with potentially improved performance. More recently, machine learning based approaches have gained attention. These techniques leverage the power of algorithms, such as Artificial Neural Networks or Gaussian Processes [152], to model the relationship between hyperparameters and performance. By learning from previous evaluations, these approaches can predict promising configurations and guide

the search towards better solutions. In addition to these general approaches, researchers have proposed hybrid methods that combine multiple techniques [148][153]. For example, the combination of Grid Search with local search methods, such as Gradient Descent or Simulated Annealing, can provide a balance between exhaustive exploration and fine grained optimization. Overall, the field of hyperparameter tuning has witnessed significant advancements, leading to the development of various techniques and approaches. Each approach offers distinct advantages and trade-offs in terms of search efficiency, adaptability to different problem domains, and robustness to noise or limited computational resources. Selecting an appropriate hyperparameter tuning approach depends on the specific requirements and constraints of the given problem. The primary goal of the research is to conduct a comprehensive analysis of hyperparameter tuning techniques employed for software quality prediction models. Specifically, the study aims to:

- Identify and review studies that have applied parameter tuning techniques in the development of prediction models in software quality related domains.
- Evaluate the effectiveness of parameter tuning in improving the performance of prediction models.
- Analyze the impact of parameter tuning on the performance of various learning algorithms used in software quality prediction.
- Provide guidelines and recommendations for practitioners and researchers regarding the application of parameter tuning techniques to enhance the predictive capability of software quality prediction models.

The subsequent sections of the chapter are structured as follows: Section 2 presents

the research questions that guide our systematic review and outlines the criteria used for selecting primary studies. Section 3 presents the outcomes of the selected studies and answers the research questions of the study. Section 4 provides practical guidelines and recommendations for practitioners and researchers.

5.2 Method

This systematic review adheres to the established guidelines outlined by reputable sources [83][154] and depicted in Figure 5.1. The review process commenced with the identification of the necessity for a systematic review, which was subsequently followed by the formulation of research questions based on the underlying motivations. A comprehensive search strategy was then devised to locate relevant primary studies. The data extraction phase involved extracting pertinent information from the primary studies to effectively address the research questions. Finally, data synthesis was performed to consolidate the findings and derive conclusive results for this review.

5.2.1 Identify the need for Systematic Review

The majority of classification algorithms utilized in software quality prediction models employ hyperparameters that significantly impact the performance of the predictors. A literature analysis conducted by Tantithamthavorn et al.[87] revealed that 87% of the commonly used classification algorithms for software defect prediction necessitate the configuration of at least one hyperparameter setting. Given this context, it becomes imperative to conduct a comprehensive review of studies that have focused on tuning hyperparameters of classification algorithms when constructing software quality prediction

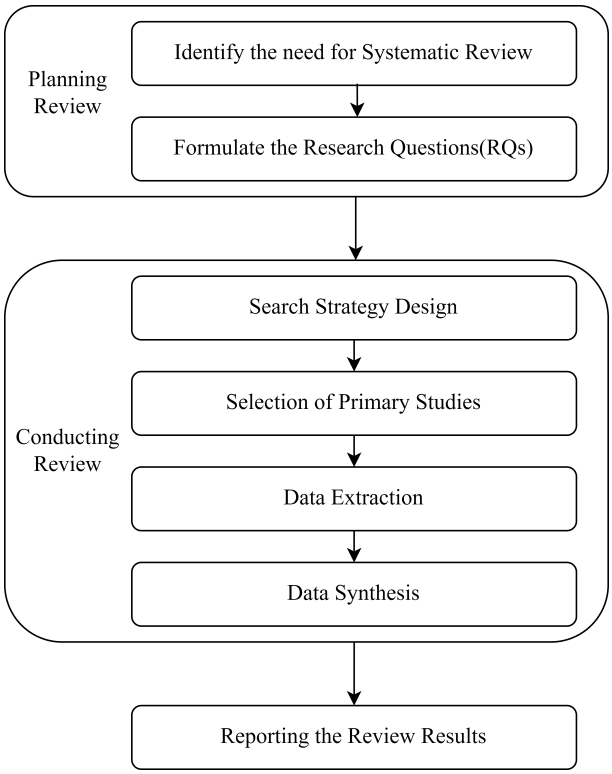

Figure 5.1: Year-wise distribution of studies

models. This review aims to investigate the following aspects:

- The performance improvement observed in tuned models compared to untuned models.
- The classification techniques that display high sensitivity to their hyperparameters.
- The commonly employed parameter optimization methods within the context of software quality prediction models.
- The associated overheads and complexities involved in the parameter tuning process.

- The prevalence and popularity of parameter tuning in software quality prediction models.

5.2.2 Research Questions

The primary objective of this review is to investigate the utilization of parameter tuning techniques in the construction of software prediction models and examine their influence on model performance. The motivations outlined in section 5.1 serve as the foundation for formulating the research questions that guide this study. Table 1 presents a comprehensive overview of the research questions addressed in this review. To address RQ1, the study meticulously analyzed the primary studies to identify the diverse software quality attributes for which parameters were tuned. Additionally, RQ2 delves into the parameter tuning techniques employed in these studies, while RQ3.1 focuses specifically on the hyperparameters of classification algorithms that underwent tuning. Furthermore, RQ3.2 explores the impact of parameter tuning on model performance, and RQ3.3 seeks to identify the most effective parameter tuning techniques employed in the reviewed studies. In pursuit of a thorough analysis, RQ4 assesses the strengths and weaknesses associated with parameter tuning in software prediction models. This examination enables us to provide valuable guidelines to researchers regarding the optimal tuning of classification algorithm parameters.

By addressing these research questions, this review aims to shed light on the current practices, challenges, and best practices surrounding parameter tuning in the development of software prediction models. The insights obtained from this analysis will serve as a valuable resource for researchers and practitioners alike, fostering improvements in the field of software engineering.

Table 5.1: Research Questions

RQ#	Research Questions	Motivation
RQ1	What are the categories of quality attributes where parameter tuning is being done?	Identify the quality attributes where parameter tuning is being applied.
RQ2	Which parameter tuning techniques have been applied in developing software quality prediction models?	Identify the techniques applied for tuning the parameters of learning algorithms in software quality models.
RQ3	What is the effect of parameter tuning on the performance of software quality prediction models?	Assess the performance of the parameter tuned software quality models vs un-tuned models.
RQ3.1	The parameters of which machine learning techniques have been tuned?	Identify the parameters reported to be appropriate for tuning.
RQ3.2	Whether the performance of quality prediction models have been improved by tuning the parameters of learning algorithms?	Investigate the improvement of performance from the results of the studies that tuned parameters.
RQ3.3	Which are the most effective parameter tuning techniques?	Identify the efficient parameter tuning techniques.
RQ4	What are the strengths and weaknesses of tuning parameters?	Determine the information about tuning parameters.
RQ5	What are the guidelines given in studies that a researcher should keep in mind while tuning the parameters?	Determine the guidelines to be followed for tuning the parameters.

5.2.3 Search Strategy and Study Selection

The objective of this study is to conduct a comprehensive review of parameter tuning techniques applied in software quality prediction models. Specifically, the focus of this

review is on the following software attributes:

1) Defect proneness

2) Maintenance proneness

3) Effort estimation

4) Reliability

To ensure a comprehensive selection of primary studies, a meticulous search strategy was devised. Synonyms and alternate terms associated with the aforementioned software attributes, prediction, and tuning were identified from the existing literature. These terms were combined using Boolean expressions, utilizing ”OR” to merge similar terms and ”AND” to combine the main search terms. The resulting search string used for the selection of primary studies is as follows: Software AND (fault OR defect OR bug OR error OR vulnerability OR change OR maintenance OR effort OR quality OR reliability) AND (proneness OR prone OR prediction OR probability) AND ((Parameter OR Hyperparameter) AND (tuning OR optimization OR selection OR determination)) The search was conducted across several reputable digital libraries, including:

1) IEEE Xplore

2) Science Direct

3) ACM Digital Library

4) Springer Link

5) Wiley Online Library

6) Google Scholar

7) Web of Science

Executing the search string on these electronic databases enabled the identification of relevant studies. In addition, a thorough examination of the references cited within these studies was conducted to identify any further pertinent research. Furthermore, select studies analyzing parameter tuning in the domain of software engineering were also considered to provide additional insights. Following a meticulous evaluation of the identified studies, a comprehensive set of 31 studies was selected as primary studies. These studies were deemed to meet the rigorous criteria outlined in the research, ensuring their relevance, credibility, and suitability for inclusion in this review.

5.2.4 Data Extraction and Data Synthesis

To ensure systematic data collection and effectively address the formulated research questions, a comprehensive data extraction form was meticulously designed. The data extraction form encompasses the following fields:

- Title of the study
- Names of the author(s)
- Publication year
- Publication details

- Datasets
- Quality attribute
- Learning/Classification algorithm(s)
- Supplementary algorithm(s) employed
- Parameters tuned/optimized
- Parameter tuning technique(s)
- Observed performance improvement upon parameter tuning
- Strengths and weaknesses of the tuning techniques
- Overall results of the study.

Each primary study underwent a thorough review process to extract relevant data as per the structured form, ensuring the accurate representation of key information. The extracted data was then meticulously recorded and organized within a spreadsheet for further analysis and synthesis.

The collected data derived from the primary studies serves as the foundation for formulating comprehensive responses to the research questions. Through a meticulous process of data synthesis, involving the summarization of pertinent facts and figures, the collected information is distilled and analyzed to generate meaningful insights and findings. This synthesis enables a comprehensive understanding of the parameter tuning techniques employed in software quality prediction models and their impact on model performance.

By following this rigorous data extraction and synthesis process, the study ensures the reliability and integrity of the findings, providing valuable insights into the effectiveness of parameter tuning in the context of software quality prediction models.

5.3 Results

This section encompasses the presentation of the results derived from the selected studies included in this systematic review. Firstly, we provide a concise overview of the chosen studies, outlining their key characteristics and contributions. Subsequently, we meticulously address each research question, drawing upon the findings extracted from the selected studies to provide comprehensive answers. Furthermore, we engage in a detailed discussion and interpretation of the results, aiming to derive meaningful conclusions and insights from the collected data.

5.3.1 Description of Primary Studies

A total of 31 studies were identified wherein parameter tuning techniques were applied in the construction of software prediction models. The details of these selected studies are presented in Table 5.2, offering comprehensive insights into the methodologies and outcomes of each study. To provide a temporal perspective, Figure 5.2 illustrates the distribution of these studies from the year 2010 to mid-2023. The graphical representation highlights that only a limited number of studies have employed parameter tuning techniques within this timeframe. Notably, the majority of these studies predominantly utilized machine learning algorithms as the foundation for their prediction models. The analysis of this distribution showcases the relatively recent emergence of parameter tuning techniques

in the realm of software prediction models.

Table 5.2: Selected Primary Studies

Study#	Paper	Reference#		Study#	Paper	Reference#
PS1	Oliveira (2010)	[155]		PS17	Medapati (2013)	[156]
PS2	Song (2013)	[157]		PS18	Jin (2014)	[158]
PS3	Arcuri (2013)	[159]		PS19	Zhaoa (2015)	[160]
PS4	Tantithamthavorn (2016)	[161]		PS20	Malhotra (2018)	[162]
PS5	Fu (2016)	[163]		PS21	Ma (2019)	[164]
PS6	Fu (2016a)	[165]		PS22	Öztürk (2019b)	[152]
PS7	Osman (2017)	[166]		PS23	Villalobos-Arias (2019)	[167]
PS8	Yang (2018)	[46]		PS24	Khan (2020)	[168]
PS9	Qu (2018)	[147]		PS25	Li (2020)	[17]
PS10	Agrawal (2018)	[169]		PS26	Lakra (2021)	[170]
PS11	Hosni (2018)	[171]		PS27	Yang (2021)	[148]
PS12	Xia (2018)	[172]		PS28	Tameswar (2022)	[173]
PS13	Kudjo (2019)	[154]		PS29	Nevendra (2022)	[113]
PS14	Öztürk (2019)	[174]		PS30	Lee (2022)	[175]
PS15	Minku (2019)	[176]		PS31	Labidi (2023)	[177]
PS16	Qin (2011)	[178]				

5.3.2 RQ1: What are the categories of quality attributes where parameter tuning is being done?

While the primary focus of this systematic review is software quality, we have also incorporated studies on effort prediction due to the shared factors that influence their respective prediction models. Figure 5.3 shows the percentage of quality attributes where

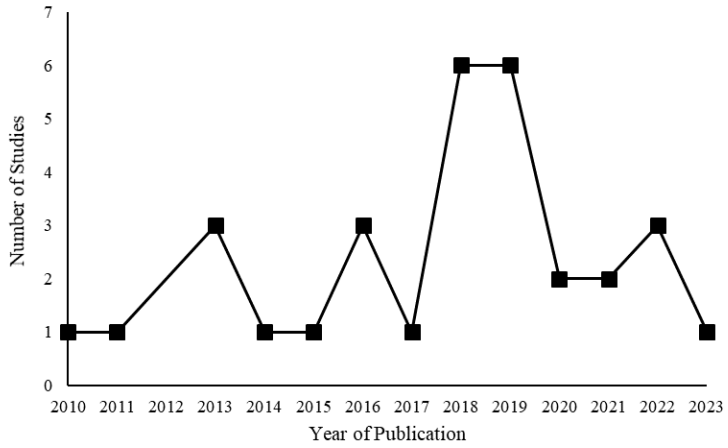

Figure 5.2: Year-wise distribution of studies

tuning is employed in the primary studies. Table 5.3 provides a comprehensive overview of the attributes herein the parameters of learning algorithms have been optimized in the selected studies. Notably, a closer examination of the table reveals a limited number of studies in each category where parameter tuning techniques have been employed. Consequently, it becomes imperative to evaluate the impact of parameter tuning on the performance of prediction models within these specific categories.

Table 5.3: Quality Attributes where parameter tuning is being done

Attribute	No. of Studies	Studies
Defect	15	PS4, PS6, PS7, PS8, PS9, PS10, PS13, PS14, PS20, PS22, PS24, PS25, PS28, PS29, PS30
Effort	7	PS1, PS2, PS11, PS12, PS15, PS23, PS31
Reliability	5	PS16, PS18, PS19, PS21, PS27
Maintenance	2	PS17, PS26
Generic software analytics	2	PS3, PS5

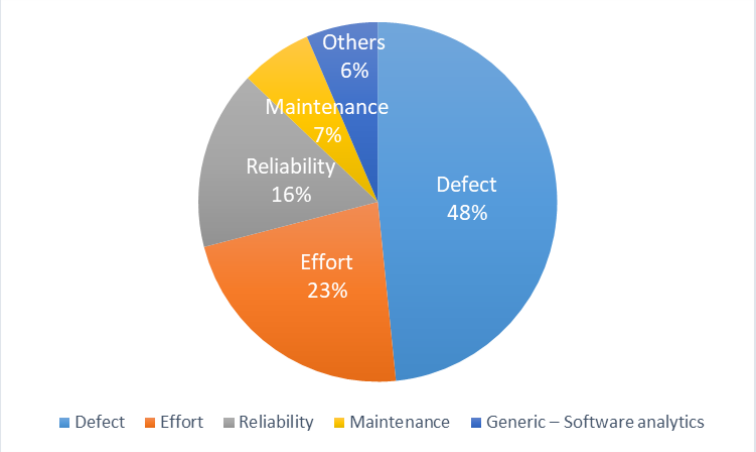

Figure 5.3: Percentage of Quality Attributes considered in Studies

5.3.3 RQ2: Which parameter tuning techniques have been applied in developing software quality prediction models?

Within the selected studies, a diverse range of parameter tuning techniques have been employed by researchers, as highlighted in Table 5.4. Figure 5.4 provides a graphical representation showcasing the distribution of studies utilizing different parameter tuning techniques. Among the various techniques employed, it is noteworthy to mention that Multisearch and Caret, which are default parameter tuning options provided by Weka [179] and R Caret [3] respectively, employ the Grid search technique. Considering Multisearch and Caret as variants of Grid search, the number of studies utilizing the Grid search technique amounts to 13, making it the most widely utilized parameter tuning technique within the selected studies. Following Grid search, the differential evolution technique is another prominent parameter tuning approach employed by researchers. Additionally, genetic algorithms and its variants have been utilized as tuning techniques in several

studies, demonstrating their efficacy in optimizing the performance of prediction models.

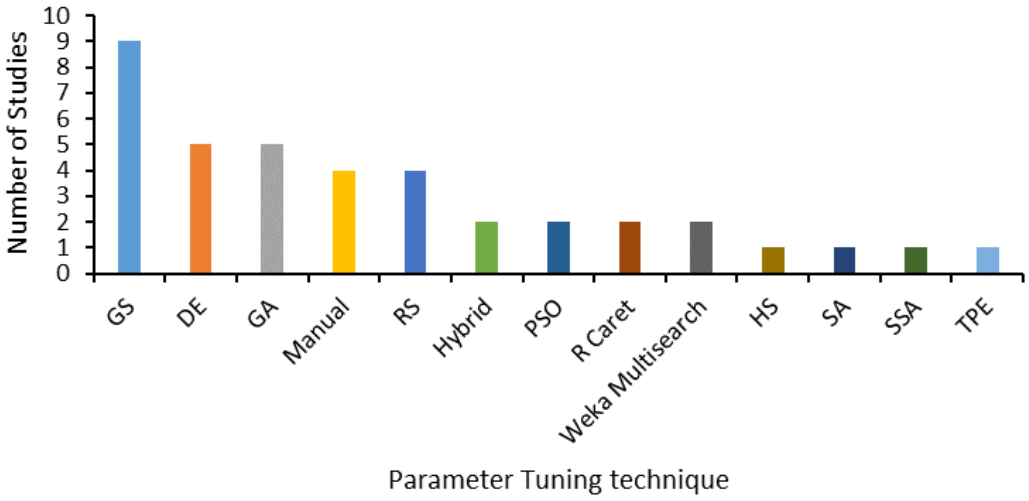

Figure 5.4: Studies using different parameter tuning techniques

Table 5.4: Parameter Tuning Techniques applied in prediction models

Technique	No. of Studies	Studies
Grid Search (GS)	9	PS6, PS7, PS11, PS14, PS21, PS22, PS26, PS29, PS31
Differential Evolution (DE)	5	PS5, PS6, PS10, PS12, PS20
Genetic Algorithm (GA) and its variations	5	PS1, PS17, PS18, PS19, PS28
Manual	4	PS2, PS3, PS13, PS15
Random Search (RS)	4	PS6, PS22, PS23, PS29
Hybrid	2	PS27, PS28
Particle Swarm Optimization (PSO)	2	PS16, PS27
R Caret	2	PS4, PS8
Weka Multisearch	2	PS7, PS9
Harmony Search	1	PS30
Simulated Annealing (SA)	1	PS20
Sparrow Search Algorithm (SSA)	1	PS27

Table 5.4 continued from previous page

Technique	No. of Studies	Studies
Tree-of-Parzen Estimators (TPE)	1	PS25

5.3.4 RQ3: What is the effect of parameter tuning on the performance of software quality prediction models?

This section presents the effect of parameter tuning on the performance of the predictors. This re-search question has sub questions and are addressed in different sub sections as follows.

5.3.4.1 RQ3.1: What are the learning techniques whose parameters have been tuned?

Table 5.5 presents the parameters of which learning techniques have been optimized. Support vector machines, k-Nearest neighbours, Random forest, Neural Networks, Classification and Regression trees, and Decision trees are widely used in the studies.

Table 5.5: Learning algorithms where parameters are optimized

Learning Algorithm	No of Studies	Studies
Support Vector Machines (SVM)	16	PS1, PS4, PS7, PS9, PS11, PS13, PS14, PS16, PS18, PS19, PS21, PS23, PS24, PS25, PS29, PS31
Random Forest (RF)	10	PS4, PS5, PS6, PS9,PS13, PS14, PS20, PS24, PS25, PS29
k-Nearest Neighbours (KNN)	8	PS2, PS4, PS7, PS9, PS11, PS13, PS24, PS25
Classification And Regression Trees (CART)	7	PS4, PS5, PS6, PS12, PS20, PS24, PS25

Table 5.5 continued from previous page

Learning Algorithm	No of Studies	Studies
Decision tree (DT)	7	PS4, PS9, PS11, PS13, PS29, PS30, PS31
Multi-layer perceptron (MLP)	7	PS1, PS2, PS4, PS11, PS13, PS25, PS29
Adaptive Boosting (AdaBoost)	4	PS4, PS22, PS24, PS25
Naive Bayes (NB)	4	PS4, PS9, PS24, PS25
Neural Network	4	PS8, PS22, PS28, PS31
Ridge regression (RR)	3	PS23, PS25, PS29
C5.0	2	PS4, PS22
Gradient Boosting Machine (GBM)	2	PS4, PS29
Bagging	1	PS2
Correlation percentile (CP)	1	PS23
DBSCANfilter	1	PS25
Deep Transfer Biclustering (DTB)	1	PS25
Domain-Specific Bias Focusing (DSBF)	1	PS25
Extra Tree Regressor	1	PS29
eXtreme Gradient Boosting Tree (xGBTree)	1	PS4
Flexible Discriminant Analysis (FDA)	1	PS4
Generalized linear and Additive Models Boosting (GAMBoost)	1	PS4
Generalized Partial Least Squares (GPLS)	1	PS4
Genetic algorithm	1	PS3
Hierarchical Agglomerative clustering	1	PS17
Huber Regressor	1	PS29
k-medoids algorithm	1	PS17
Lasso Regression (LASSO)	1	PS29
Linear discriminate analysis (LDA)	1	PS24
Linear Regression	1	PS29
Logistic Model Trees (LMT)	1	PS4
Logistic Regression (LR)	1	PS29
Logistic Regression Boosting (LogitBoost)	1	PS4
M5P algorithm (Model trees)	1	PS1
MARS	1	PS4
Penalized Discriminant Analysis (PDA)	1	PS4
Regression Tree(RT)	1	PS2
Ripper classifier (Ripper)	1	PS4

Table 5.5 continued from previous page

Learning Algorithm	No of Studies	Studies
SMOTE	1	PS10
Software Reliability Growth Model - Goel-Okumoto (G-O) model	1	PS27
Square Loss Gradient Boost (SLG)	1	PS22
Transfer Component Analysis (TCA)	1	PS25
Universal	1	PS25
Variance threshold (VT)	1	PS23
Where-based learner	1	PS5
XGBoost Regression	1	PS29

5.3.4.2 RQ3.2: Whether the performance of quality prediction models have been improved by tuning the parameters of learning algorithms?

The primary objective of parameter tuning is to enhance the performance of prediction models. In this regard, the impact of tuning the parameters of learning algorithms on the performance of prediction models, as observed in the selected studies, has been meticulously analyzed and consolidated in Table 5.6. The results derived from most of the studies consistently demonstrate a noticeable improvement in the performance of software quality prediction models following parameter tuning. In an effort to provide comprehensive insights, we have also sought to ascertain the statistical characteristics of the results obtained from the primary studies. While a variety of performance measures were employed across the selected studies, certain commonly used measures such as accuracy, area under the receiver operating characteristics curve (AUC), precision, sensitivity, and F-measure were identified. To facilitate comparison, Table 5.7 presents the minimum (Min.), maximum (Max.), mean, median, and standard deviation (std.) values of these

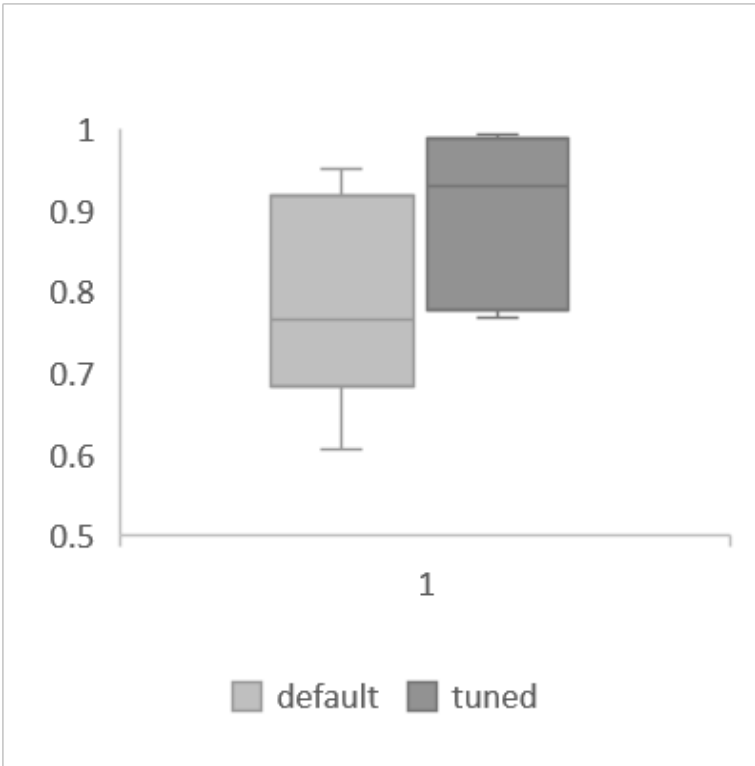

Figure 5.5: Boxplot of Accuracy Values

performance measures for models constructed with both default parameters and tuned parameters. To further illustrate the impact of parameter tuning on performance, box plots for accuracy and precision are depicted in Figure 5.5 and Figure 5.6 respectively. These visual representations clearly demonstrate the reasonable improvements achieved through parameter tuning, underscoring its efficacy in enhancing the overall performance of software quality prediction models.

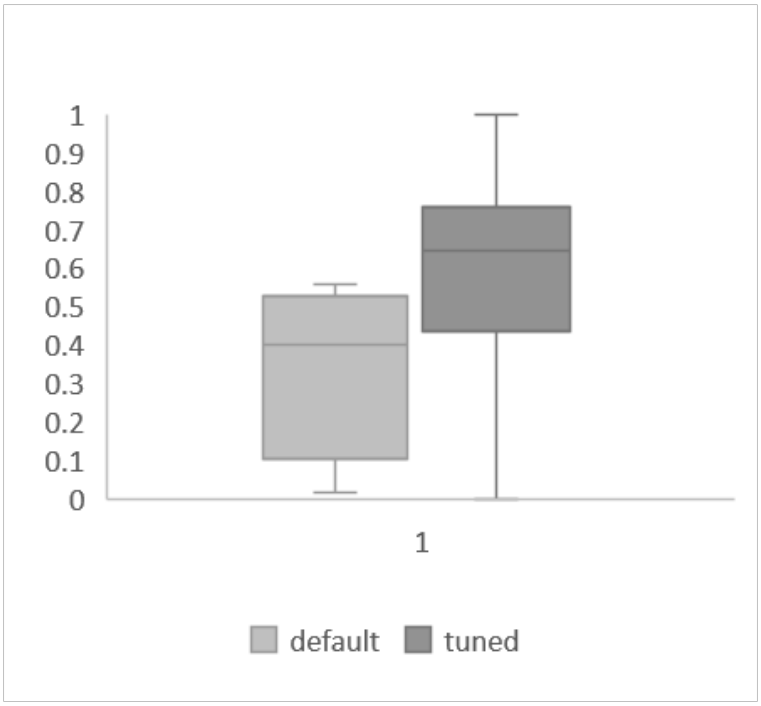

Figure 5.6: Box plot of Precision Values

Table 5.6: Improvement of performance of prediction models - study wise

Study#	Improvement of Performance
PS1	The performance of support vector regression (SVR), multi-layer perceptron (MLP), and model trees (M5P algorithm) significantly improved.
PS2	Regression Trees (RTs) and Bagging+RTs show limited sensitivity to different parameter settings. However, it is recommended to tune the parameters for improved performance. MLPs and Bagging+MLPs exhibit excellent performance but are highly sensitive to their parameter settings, including the initial configurations. k-NN demonstrates low sensitivity to its parameter settings, except when k equals one (1-NN), which consistently yields poor performance across all datasets.

Table 5.6 continued from previous page

Study#	Improvement of Performance
PS3	Different parameter configurations result in significant performance variation. Although default parameter settings exhibit relatively satisfactory performance, they are suboptimal for specific problem instances.
PS4	Caret enhances the AUC performance by a substantial 40 percentage points. Additionally, Caret continues to enhance cross-context defect prediction models' performance by up to 30 percentage points.
PS6	Random search (RS) and Differential Evolution (DE) both enchance classifiers performance in terms of precision and F-measure, respectively.
PS5	Tuning seldom results in performance deterioration and frequently leads to significant improvement, with precision increasing from 0 to 60%.
PS7	The prediction accuracy is enhanced by as much as 20% in Instance-based learning with parameter k (IBk) and up to 10% in Support Vector Machines (SVM). IBk exhibits higher sensitivity to hyperparameter tuning compared to SVM. Hyperparameter tuning not only has the potential to improve prediction accuracy but also influences the relative performance of different machine learning models.
PS8	Specificity, accuracy and AUC are all improved by using Caret-optimized setting.
PS9	By using hyper parameter optimization, the performance can be increased by 34.63 percentage points at most. Moreover, hyper parameter optimization is most effective for IBk.
PS10	Differential Evolution can significantly enhance predictive performance, with improvements of up to 60% in the area under the curve, by tuning SMOTE's hyperparameters.
PS11	Particle Swarm Optimization (PSO) and Grid Search(GS) generate more accurate results than Uniform Configuration(UC-WEKA) (default parameters settings of Weka tool).
PS12	CART optimized by DE8 or MOEA/D achieves top-ranked results (in 8/9 cases).
PS13	k-NN and J48 demonstrated enhanced prediction accuracy. Random forest classifier yielded high precision, recall and accuracy values.

Table 5.6 continued from previous page

Study#	Improvement of Performance
PS14	The maximum tree depth (interaction.depth) plays a vital role in improving accuracy when utilizing tree-based algorithms. Hyperparameter tuning proves advantageous in defect prediction when employing SVM, regardless of prediction settings.
PS15	LogLR does not require any parameters, while RTs and BagRTs are recognized to be relatively unaffected by their hyperparameters in the domain of online SEE.
PS16	PSO-SVM slightly better performance than that of SVM.
PS17	Tuned parameters give better results and the precision and recall values are higher.
PS18	(Improve Estimation of Distribution Algorithms) IEDA-SVR obtained the better prediction results.
PS19	The utilization of Analytic Selection (AS) significantly enhances GA optimization, resulting in improved accuracy, increased robustness, and faster convergence.
PS20	In approximately 75% of the datasets, the tuned model exhibits superior or comparable performance to the untuned model. Hence, in general, tuning consistently improves performance, except for rare instances.
PS21	Parameter tuning leads to improved accuracy in the prediction results.
PS22	Parameter Optimization has increased AUC values by 0.2.
PS23	Random search improves the performance of SVR models to a maximum of 0.227.
PS24	SVM-RBF and Linear discriminate analysis(LDA) more sensitive.
PS25	Automated parameter optimization yields substantial improvement in prediction, with up to 77% showing a significant effect size according to Cohen's rule.
PS26	For the QUES dataset, the average improvements in R-squared, MAE, and RMSLE measures were 20.24%, 12.26%, and 30.28%, respectively. Conversely, the UIMS dataset exhibited average improvements of 6.27%, 15.71%, and 16.39% in R-squared, MAE, and RMSLE measures, respectively.
PS27	Tuning demonstrates significant enhancements in both convergence speed and stability of the reliability models.

Table 5.6 continued from previous page

Study#	Improvement of Performance
PS28	Deep Neural Network - combined with Genetic Algorithm and Coral Reefs Optimization (DNN-GA-CRO) achieved highest accuracy of 96.67% and average F1-score of 0.92, while Deep Neural Network combined with Genetic Algorithm and Cuckoo Search (DNN-GA-CS) with highest accuracy of 93.78% and average F1-score of 0.90.
PS29	Hyperparameter optimization greatly enhances prediction performance for MLP Regression (MLPR), Lasso, Decision Tree Regression (DTR), Hubber, and Support Vector Regression (SVR), resulting in improvements of 16.96%, 8.31%, 8.16%, 6.01%, and 5.22% respectively. However, linear regression does not demonstrate sensitivity to hyperparameters. Grid search (GS) improves performance by 4.42%, while random search (RS) improves it by 3.36%. Even non-significant classifiers like MLP exhibit substantial changes in their rankings after hyperparameter optimization in the domain of software defect prediction (SDP). Logistic regression achieves the highest ranking in terms of hyperparameter optimization.
PS30	(1) Harmony Search (HS) outperforms the current state-of-the-art hyperparameter tuning methods in terms of performance. (2) The most influential factors on performance are feature selection, followed by class weight, normalization, and model hyperparameters. (3) In general, HS demonstrates superior performance compared to traditional tuning techniques such as Grid Search, Random Search, Tabu Search, and Genetic Algorithm.
PS31	Grid search (GS) tuning enhances the performance of both individual and stacking models by swiftly and precisely identifying optimal hyperparameter settings.

5.3.4.3 RQ3.3: What are the most effective parameter tuning techniques?

The majority of the selected studies focused on the application of a single parameter tuning technique and did not conduct comparative analyses of different tuning techniques. However, a few studies stand out for their efforts in directly comparing the performance

of various tuning techniques. For instance, PS6 conducted a comparative study between differential evolution and grid search for parameter tuning in defect predictors. The findings of this study demonstrated that differential evolution outperformed grid search in terms of optimizing the performance of defect predictors. Similarly, PS11 investigated the impact of grid search and particle swarm optimization on parameter tuning. The results of this study revealed that both techniques exhibited generally positive effects on parameter optimization, without significant differences in performance. Furthermore, the results presented in PS20 indicated that differential evolution outperformed simulated annealing in terms of performance improvement. This finding strengthens the evidence supporting the efficacy of differential evolution as a superior parameter tuning technique. The results of PS30 indicate that Harmony Search shows better performance compared with the traditional optimization methods (Grid Search, Random Search, Tabu Search and Genetic Algorithm).

Among the various parameter tuning techniques, grid search, random search, genetic algorithm and differential evolution emerged as particularly popular choices among researchers, with multiple studies incorporating these techniques into their experimentation.

Table 5.7: Values of performance measures of models with default and tuned parameters settings

Performance Measure	Parameter Setting	Min.	Max.	Mean	Median	Std.
Accuracy	Default	0.607	0.95	0.771	0.765	0.124
	Tuned	0.767	0.992	0.881	0.929	0.102
AUC	Default	0.5	0.5	0.5	0.5	0
	Tuned	0.5358	0.5619	0.549	0.549	0.018
	Default	0.018	0.556	0.337	0.399	0.225

Table 5.7 continued from previous page

Performance Measure	Parameter Setting	Min.	Max.	Mean	Median	Std.
Precision	Tuned	0.5784	1	0.591	0.646	0.326
F-Measure	Default	0.3483	0.55	0.447	0.444	0.1
	Tuned	0.4965	0.59	0.55	0.563	0.048

5.3.5 RQ4: What are the strengths and weaknesses of tuning parameters?

The findings across the majority of studies consistently demonstrate a substantial minimum 30 percent improvement in the performance of the predictors following parameter tuning. Notably, even the learning algorithms that initially performed poorly exhibited significant performance enhancements when their parameters were properly tuned, often surpassing the performance of the top-performing algorithms. This highlights the potential for parameter tuning to effectively address the limitations of underperforming models and unlock their full predictive capabilities.

However, it is important to acknowledge the potential weaknesses associated with parameter tuning techniques. Studies have identified two key concerns:

- *Additional Computational Cost:* The pursuit of optimal parameter settings requires additional computational resources and time. Tuning parameters often involves exhaustive search or optimization algorithms, which can significantly increase the computational burden of model training and evaluation. Researchers must carefully consider the trade-off between the potential performance improvement and the computational cost incurred.

- *Risk of Overfitting:* The tuning process introduces the risk of overfitting, wherein the model becomes excessively tailored to the training data and loses its ability to generalize well to new, unseen data. Fine-tuning parameters can result in models that exhibit high accuracy on the training set but perform poorly on unseen data. This highlights the need for cautious parameter tuning to strike a balance between model complexity and generalizability.

5.3.6 RQ5: What are the guidelines given in studies that a researcher should keep in mind while tuning the parameters?

The research findings from various studies have shed light on several important aspects related to parameter tuning in software prediction models:

- Different parameter settings exhibit significant variance in performance, indicating that default parameter settings, while relatively satisfactory, are far from optimal for individual problem instances. Tuning parameters can lead to improvements on average, but still fall short of achieving optimality for specific instances.

- A substantial majority (87%) of the most commonly used classification techniques, as indicated by 26 out of 30 techniques, require at least one parameter setting. This underscores the criticality of selecting optimal parameter settings as an important experimental design choice for defect prediction models.

- Parameter tuning has the potential to alter the comparative rankings of data mining algorithms, emphasizing its impact on model performance evaluation.

- Certain algorithms, such as decision tree, support vector machine and random forest, display high sensitivity to parameter optimization, suggesting the need for careful tuning to achieve optimal results.
- Neural network based algorithms requires a high-performance computing system to effectively handle the tuning process, indicative of its computational demands.

Despite the significant findings highlighting the importance of parameter tuning, a significant number of researchers and practitioners still overlook the tuning of classification algorithm parameters in software prediction models. This reluctance can be attributed to the following factors:

- Constraint of time
- Computational overhead
- Unware of its significance

To address this gap, we propose guidelines for software researchers and practitioners when applying parameter tuning techniques:

- *Assess the sensitivity of the classification algorithm to its parameters:* Different algorithms exhibit varying levels of sensitivity to their hyperparameters. Understanding this sensitivity is crucial for determining the impact of parameter settings on model performance.
- *High sensitivity:* If the classification technique is highly sensitive to its parameters, parameter tuning becomes imperative to achieve optimal or near-optimal settings. In

such cases, researchers should employ the most suitable parameter tuning technique to fine-tune the hyperparameters.

- *Medium sensitivity:* If the classification technique demonstrates moderate sensitivity to its parameters, the impact of parameter settings on model performance is moderate. If practitioners face constraints in tuning the parameters, default parameter tuning methods provided by data mining tools or packages can be considered. For example, the Weka tool offers Multisearch as its default parameter tuning method, while the Caret package in R provides automatic parameter tuning.

- *Low sensitivity:* In instances where the classification technique exhibits low sensitivity to its parameters, the impact of parameter settings on model performance is minimal. In such cases, practitioners may choose to disregard parameter tuning if they face constraints or limitations.

To facilitate decision making regarding parameter tuning, Figure 5.7 presents a flowchart depicting the process based on the sensitivity of the classification technique.

These guidelines aim to support software researchers and practitioners in making informed decisions when it comes to parameter tuning, considering the specific characteristics and sensitivity of the classification algorithms employed. By following these guidelines, researchers and practitioners can enhance the performance of software prediction models and optimize their resource allocation effectively.

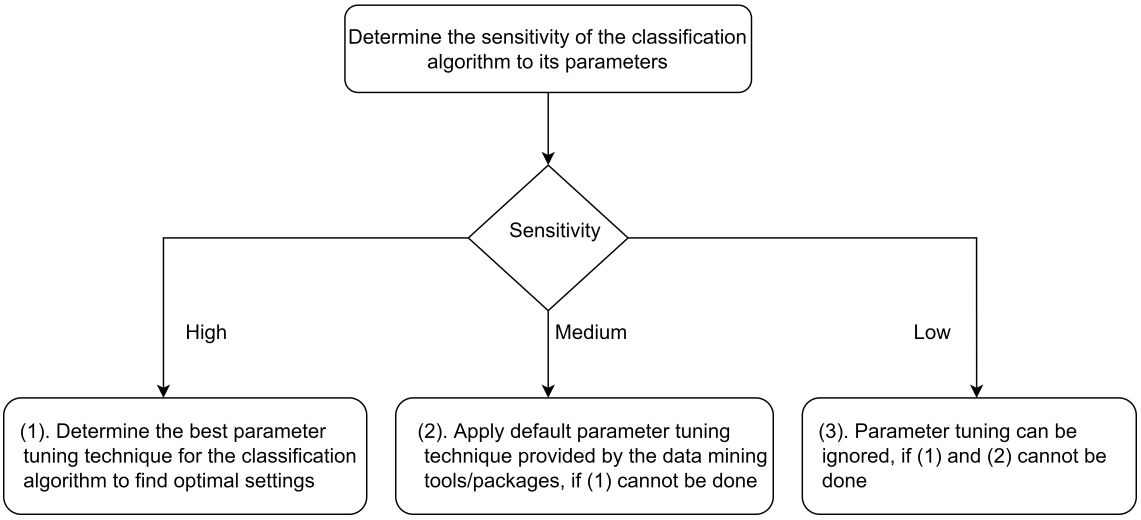

Figure 5.7: Flow chart to determine tuning method

5.4 Discussion

This study conducted a meticulous systematic literature review to examine the utilization of parameter tuning techniques in software quality prediction models, specifically focusing on defect, maintenance, reliability, and effort attributes of the software. The key objectives of this study were as follows: First, we performed an extensive search within digital libraries and identified 31 primary studies that implemented parameter tuning techniques within the context of software quality prediction models. Second, we carefully extracted and synthesized data from these primary studies. We summarized the characteristics of the primary studies based on various quality attributes, parameter tuning techniques, classification algorithms, and their respective hyperparameters. Third, we conducted a comprehensive analysis of the primary study results to evaluate the impact of parameter tuning on the performance of software quality prediction models. Additionally, we

analyzed studies that employed multiple parameter tuning techniques to determine the most effective approach. Fourth, we provided a thorough examination of the strengths and weaknesses associated with tuning the hyperparameters of classification algorithms. Finally, we presented guidelines and recommendations that software practitioners should consider when tuning parameters for their prediction models.

The main findings derived from the selected primary studies are as follows:

- A limited number of studies have specifically addressed parameter tuning settings in the realm of software quality prediction, resulting in untuned models that are far from optimal in terms of performance.
- Tuned models consistently exhibited improved prediction capabilities and demonstrated stability comparable to untuned models.
- Grid search, Differential evolution, Genetic algorithm-based and hybrid parameter tuning techniques emerged as the most commonly employed and effective methods.
- The parameters of classification algorithms such as Support Vector Machine, k-nearest neighbor, Random Forest, Neural Networks, Classification and Regression Trees (CART), and Random Forest Classification were frequently subjected to tuning.
- Notably, Neural Networks, Instance-based Learning with parameter k (IBk), Support Vector Machine, Decision Tree, Random Forest and Logistic Regression exhibited high sensitivity to parameter tuning. Linear Regression, Regression Tress (RTs) and Bagging+RTs exhibited less sensitivity to hyperparameters.

- Parameter tuning was observed to significantly enhance the performance of underperforming classification techniques, leading to notable changes in their ranking.

Based on the results obtained, we strongly recommend software practitioners to adopt parameter tuning techniques when constructing their prediction models. For practitioners facing time constraints, utilizing default parameter tuning methods provided by tools like Weka-Multisearch and R-Caret can serve as a valuable starting point. Furthermore, we urge researchers to conduct comparative studies to further evaluate the effectiveness of different parameter tuning techniques. Additionally, the exploration of parameter tuning techniques on a broader range of classification algorithms is encouraged to expand the understanding of their impact.

By implementing these recommendations and incorporating parameter tuning techniques, software practitioners and researchers can enhance the performance and reliability of their software quality prediction models, thereby advancing the field of software engineering.

Chapter 6

Addressing Imbalanced Data in Software Defect Prediction: A Focus on Neural Networks

6.1 Introduction

In today's era of increased automation across industries, the size and complexity of software systems have witnessed exponential growth [180]. However, building large software systems without defects within a given time and budget constraints remains a challenging task for both managers and developers. Software defect prediction (SDP) models have gained prominence as a solution for early identification of defect-prone software components, enabling optimal allocation of resources for thorough testing of these modules[181]. Machine learning algorithms have proven effective in building defect

prediction models, surpassing traditional statistical models [31]. Neural Networks have gained popularity due to advancements in computing power and storage. A overview of Artificial Neural Networks is given Chapter 2.

Nevertheless, the performance of SDP models is frequently impeded by the presence of imbalanced data distributions in the target variable categories. Imbalanced data refers to ”a situation where one class, known as the majority class, has significantly more data points than the other class” [182]. This data skewness poses difficulties for most classification techniques, which tend to be biased towards learning and identifying majority class data points, leading to incorrect classification of minority class data points [183] [101]. In the context of software defect datasets, the number of defective classes (minority class) is often considerably lower compared to the number of non-defective classes (majority class). Consequently, standard machine learning models that assume equal costs for misclassifying defective and non-defective classes (false negatives and false positives) often perform poorly for the minority class [128]. Researchers have extensively explored the issue of class imbalance in the past decade. Various methods have been proposed to tackle class imbalance, which can be generally classified into two categories: data-level and algorithm-level techniques [128] [184]. Data-level methods involve manipulating the training data through resampling techniques [185]. These techniques aim to address class imbalance independently of the classification technique by rebalancing the class distribution in the training set. On the other hand, algorithm-level methods modify the classification techniques to handle class imbalance directly, without altering the underlying dataset [186]. This chapter investigates the application of both data-level and algorithm-level approaches to handle the issue of imbalanced data in NN-based SDP models.

6.2 Application of Data Resampling Techniques

This approach investigates the use of 12 different data resampling techniques, four oversampling techniques, six undersampling techniques, and two hybrid techniques (a combination of undersampling and oversampling) applied on the six distinct defect datasets of open-source java-based systems. Thus, a total of 6 (datasets) x 13 (12 resampling techniques + 1 no resampling) = 78 defect prediction models were developed using ten-fold cross validation and their performance is assessed with three reliable and consistent performance metrics, namely AUC, G-Mean and Balance. By evaluating the performance of 78 different models, the study aims to determine the most effective resampling technique for improving ANN-based SDP models.

The research questions (RQs) formulated to achieve the objectives of the study are:

RQ1. How well do ANN-based SDP models perform on imbalanced datasets.?

RQ2. Can the predictive capability of ANN-based SDP models be improved by different data resampling methods.?

RQ3. Does the performance of the techniques based on combination of over-sampling and under sampling compared to performance of individual oversampling and under sampling techniques.?

RQ4. Which data resampling technique outperform the other techniques in ANN-based SDP models.?

6.2.1 Proposed Approach

This study aim to address the issue of imbalanced data in ANN-based software defect prediction models. To achieve this, various data sampling techniques, including oversampling, undersampling, and hybrid techniques on distinct defect datasets of open-source Java-based systems are applies. The proposed approach of this study is presented in Figure 1.1.

6.2.2 Datasets

The software projects validated in the study are six open source java applications - ant, camel, ivy, jedit, log4j and prop. The description and characteristics of these datasets are given in Chapter 2.

6.2.3 Data balancing techniques

In this work, the defective class is considered as the minority class while the non-defective class is considered as the majority class. The approach of data resampling techniques involves adjusting the class distribution in the dataset. This is achieved by either oversampling the minority class to increase its frequency, or undersampling the majority class to decrease its frequency. This study applied 14 different data balancing techniques from oversampling and undersampling approaches. The overview of different techniques employed in this study are presented as follows:

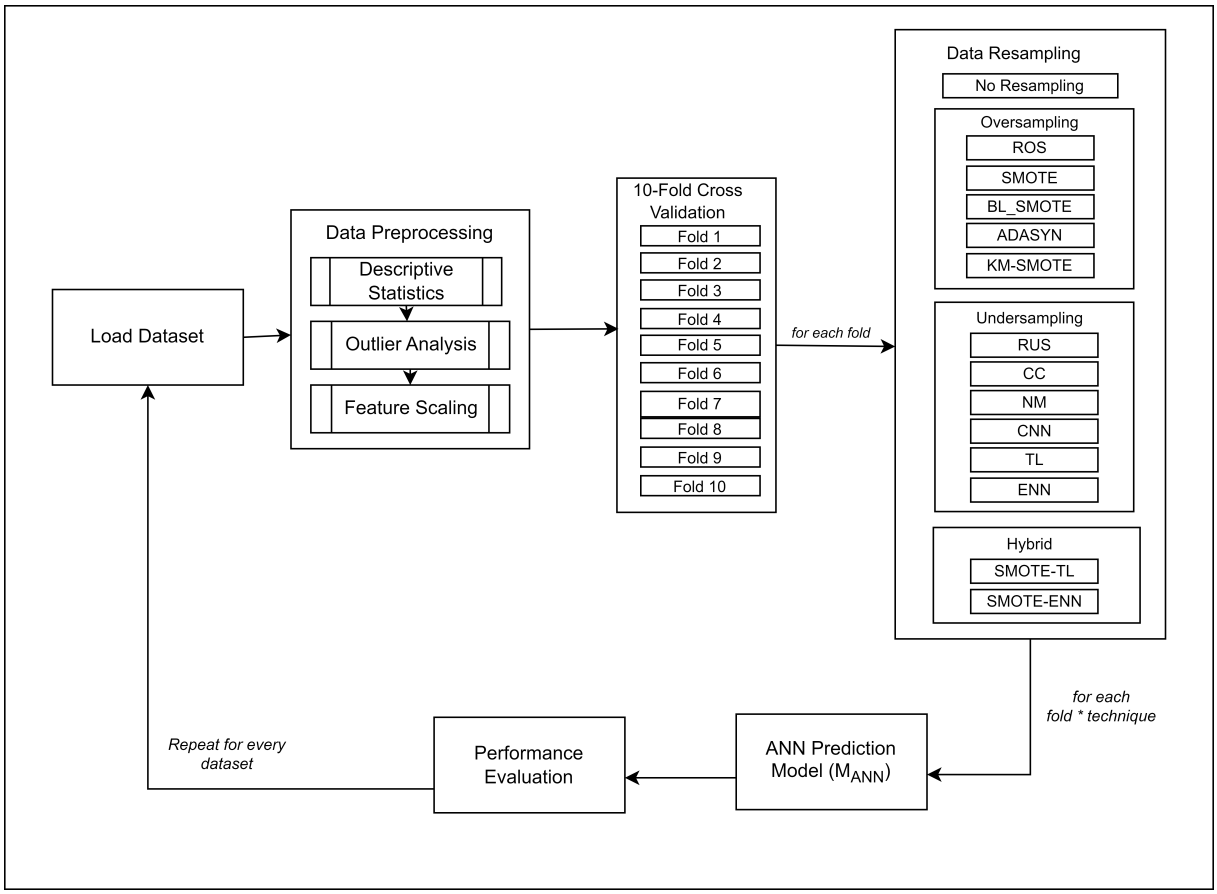

Figure 6.1: Proposed Approach for handing imbalanced data through datasampling techniques

Oversampling Techniques

- **Random Over Sampling (ROS):** This naive strategy generates new instances of minority class by randomly sampling from the current available instances with replacement [185].

- **Synthetic Minority Oversampling Technique (SMOTE):** SMOTE choose random

minority instance, then constructs feature space vectors between K nearest neighbor instances from the selected instance and generate new synthetic instances on the lines [187].

- **Borderline Synthetic Minority Oversampling Technique (BL-SMOTE):** BLSMOTE is a variant of SMOTE, that only generate synthetic instances of minority class instances near the boundary line [188].
- **Adaptive Synthetic Sampling (ADASYN):** Although similar to SMOTE, it differs in that it has a bias towards sample space points that are not located within homogenous neighborhoods [189].
- **K-Means Synthetic Minority Oversampling Technique (KM-SMOTE):** KMSMOTE involves clustering of minority class instances utilizing the k-means clustering algorithm. Additional synthetic samples are then allocated to clusters that have a sparse distribution of minority class samples [190].

Undersampling Techniques

- **Random Under Sampling (RUS):** is a simple technique delete instances of the ma-jority class randomly and uniformly [185].
- **Cluster Centroids (CC):** is a method that removes majority class instances by substituting a cluster of majority instances. This is accomplished by use of K-means algorithm to cluster majority instances. The majority class clusters are then replaced with the centroids of the N clusters, which then become the new majority class instances [191]

- **Near Miss (NM):** The NM method works by reducing the number of majority class instances in the training dataset by selecting samples from the majority class that are closest in distance to the minority class [192].
- **Condensed Nearest Neighbor Rule (CNN):** The CNN method involves the inclusion of all instances of the minority class and only incremental addition of instances from the majority class that cannot be classified correctly [193].
- **Tomek Links (TL):** In the TL method, pairs of instances are identified, one from the minority class and one from the majority class, that have the smallest Euclidean distance to each other in feature space, which are known as ”Tomek Links”. The majority class instances that are closest to the minority class are then removed [194].
- **Edited Nearest Neighbor Rule (ENN):** The ENN method involves removing instances whose class label differs from the classes of at least two out of their three nearest neighbors [195].

Combination of Over and Under Sampling Techniques

- **SMOTE with Tomek Links (SMOTE-TL):** The approach involves the use of SMOTE for oversampling and Tomek links for cleaning the dataset [196].
- **SMOTE with Edited Nearest Neighbors (SMOTE-ENN):** The approach involves the use of SMOTE for oversampling and Edited Nearest Neighbors for cleaning the dataset [185].

6.2.4 Experimental Setup

The study developed the models by programming in Python. Keras, the python deep learning API [146] is used for building the neural network. The ANN is built with the structure as presented in Table 6.1. The parameter configurations of ANN are presented in Table 6.2 and the remaining parameters are set to default. Imbalanced-learn API [197] is used for implementation of data resampling techniques and scikit-learn [61], a machine learning API is used for data preprocessing, model fitting, model evaluation and others utilities. 10-fold cross validation is applied for validating the model [198]. Fifteen datasets (one imbalanced dataset and fourteen balanced datasets generated by applying 14 different data resampling techniques) are generated from every master dataset considered for this study. Model is trained and tested on these datasets and their performance measures (AUC, GM, BL) are saved. Then the statistical tests were conducted to compare the effectiveness of these methods - Friedman test was first utilized, followed by a Wilcoxon post-hoc analysis.

Table 6.1: Structure of ANN constructed in the study

Layer	No. of Nodes	Activation Function
Input Layer	as number of independent variables	-
Hidden Layer 1	18	relu
Hidden Layer 2	15	relu
Hidden Layer 3	10	relu
Hidden Layer 4	5	relu
Output Layer	1	sigmoid

Table 6.2: Parameter configuration of ANN constructed in the study

Parameter	Value
Regularizer	L2 regularization penalty of 0.001
Optimizer	Optimizer based on RMSprop algorithm
Loss function	Poisson
Metrics	AUC, GM, BL
Epochs	10

6.2.5 Results

The study's research questions are addressed in this section by evaluating the performance of SDP models of various techniques. Tables 6.3, 6.4 and 6.5 present the values of performance metrics AUC, GM and BL respectively.

RQ1. How well do ANN-based SDP models perform on imbalanced datasets.?

To answer this RQ; we developed ANN-based SDP models on the imbalanced datasets and by applying ten-fold cross-validation. The performance of developed models is assessed using AUC, GM and BL metrics (shown in Tables 6.3, 6.4 and 6.5). The AUC values of models using Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets are 0.5414, 0.5579, 0.458, 0.4547, 0.4969 and 0.6196 respectively. From Table 6.4, the GM values of models using Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets are 53.79, 53.84, 49.9, 53.6, 54.48 and 61.83 respectively. From Table 6.5, the BL values of models using Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets are 53.52, 53.75, 49.48, 52.62, 54.26 and 61.48 respectively. A higher AUC, GM and BL values means that the performance of the

model is much better. The AUC, GM and BL values of the developed model on the six imbalanced datasets are far from satisfactory.

Table 6.3: AUC Results of SDP Models using data resampling techniques

Technique	Ant	Camel	Ivy	Jedit	Log4j	Prop
IMBALANCED	0.5414	0.5579	0.458	0.4547	0.4969	0.6196
ROS	0.639	0.6571	0.7901	0.8376	0.6496	0.7041
SMOTE	0.6182	0.621	0.7919	0.8946	0.6706	0.7034
BL-SMOTE	0.6559	0.6985	**0.8438**	0.8322	0.6874	0.719
ADASYN	0.5929	0.5886	0.7374	0.8185	0.5832	0.6652
RUS	0.6189	0.4995	0.7225	0.4876	0.455	0.4386
CC	0.4962	0.6106	0.3706	0.4463	0.5625	0.5951
NM	0.5169	0.5632	0.4647	0.4463	0.5	0.4299
CNN	0.5687	0.4939	0.4804	0.4773	0.53	0.4695
TL	0.5723	0.5658	0.4936	0.4886	0.4645	0.5914
ENN	0.6539	0.4683	0.1647	0.3847	0.5571	0.3598
SMOTE-TL	0.6298	0.6402	0.7935	0.8237	0.6831	0.7372
SMOTE-ENN	**0.7272**	**0.7316**	0.817	**0.9292**	**0.7762**	**0.843**

Table 6.4: GM Results of SDP Models using data resampling techniques

Technique	Ant	Camel	Ivy	Jedit	Log4j	Prop
IMBALANCED	53.79	53.84	49.9	53.6	54.48	61.83
ROS	61.2	62.15	77.5	78.87	62.01	64.76
SMOTE	59.03	57.91	75.91	84.62	62.7	64.96
BL-SMOTE	62.65	65.87	**79.48**	78.17	68.39	67.54
ADASYN	59.19	57.34	68.36	75.28	58.99	64.08
RUS	58.59	51.08	66.14	53.78	50	49.99

Table 6.4 continued from previous page

Technique	Ant	Camel	Ivy	Jedit	Log4j	Prop
CC	51.12	60.25	46.77	53.78	58.31	61.16
NM	54.19	58.47	47.5	54.55	56.12	50.39
CNN	55.99	50.43	48.99	54.26	56.41	50.82
TL	57.01	54.81	51.26	59.22	49.63	58.84
ENN	64.32	48.68	28.56	52.68	59.9	42.29
SMOTE-TL	61.17	60.49	77.33	78.67	65.11	68.39
SMOTE-ENN	**68.26**	**68.6**	74.21	**87.65**	**71.85**	**77.69**

RQ2. Can the predictive capability of ANN-based SDP models be improved by the application of different data resampling techniques.?

To investigate this RQ; we built ANN-based SDP models on the balanced datasets by resampling datasets using 10 data resampling techniques (ROS, SMOTE, BL-SMOTE, ADASYN, RUS, CC, NM, CNN, TL and ENN) and by applying ten-fold cross-validation. For each of 10 data resampling techniques on 6 datasets, 10 x 6 = 60 models were developed. The performance of these models are assessed using AUC, GM and BL metrics.

Table 6.3 shows AUC values of the models after developed on balanced datasets. ROS technique showed the improvement of 18%, 18%, 73%, 84%, 31% and 14% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively. SMOTE technique showed the improvement of 14%, 11%, 73%, 97%, 35%, and 14% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively. BL-SMOTE technique showed the improvement of 21%, 25%, 84%, 83%, 38% and 16% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively.

ADSYN technique showed the improvement of 10%, 6%, 61%, 80%, 17% and 7% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively. The under sam-pling techniques not showed any improvement of AUC values except for few da-tasets.

Table 6.4 shows Geometric Mean(GM) values of the models after developed on bal-anced datasets. ROS technique showed the improvement of 14%, 15%, 55%, 47%, 14% and 5% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets re-spectively. SMOTE technique showed the improvement of 10%, 8%, 52%, 58%, 15% and 5% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respective-ly. BL-SMOTE technique showed the improvement of 16%, 22%, 59%, 46%, 26% and 9% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respective-ly. ADSYN technique showed the improvement of 10%, 7%, 37%, 40%, 8% and 4% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively. The under sampling techniques not showed any improvement of GM values except for few data sets.

Table 6.5 presents Balance(BL) values of the models after developed on balanced datasets. ROS technique showed the improvement of 14%, 15%, 56%, 49%, 14% and 5% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respective-ly. SMOTE technique showed the improvement of 10%, 8%, 53%, 58%, 16% and 5% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively. BL-SMOTE technique showed the improvement of 17%, 23%, 58%, 49%, 25% and 9% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively. ADSYN technique showed the improvement of 10%, 7%, 38%, 38%, 8% and 3%

over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively. The under sampling techniques not showed any improvement of BL values except for few data sets.

In case of extremely unbalanced dataset Jedit where there are only 2% of defective classes, oversampling techniques ROS, SMOTE, BL-SMOTE and ADASYN showed improvement of AUC values of 84%, 97%, 83% and 80% respectively.

It is evident from the outcomes that there is noteworthy enchacement in performance (AUC, GM, BL) of the SDP models constructed after balancing the dataset with the oversampling techniques considered in this study. The defect prediction models developed after balancing the dataset with undersampling techniques considered in this study didn't shown any improvement of performance except for few datasets.

RQ3. Does the performance of the techniques based on combination of oversampling and under sampling compared to performance of individual oversampling and under sampling techniques.?

Two combined techniques of oversampling and under sampling studied in this work are SMOTE-TL and SMOTE-ENN. 2 x 6 = 12 defect prediction models developed for six data sets using SMOTE-TL and SMOTE-ENN. The predictive capability of these models are evaluated by AUC, GM and BL.

Table 6.3 shows AUC values of the models constructed on datasets balanced through SMOTE-TL and SMOTE-ENN. SMOTE-TL technique showed the improvement of 16%, 15%, 73%, 81%, 37% and 19% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively. SMOTE-ENN technique showed the improvement of 34%, 31%, 78%, 104%, 56% and 36% over Ant, Camel, Ivy, Jedit, Log4j and

Prop imbalanced datasets respectively.

Table 6.4 shows GM values of the models after developed on datasets balanced through SMOTE-TL and SMOTE-ENN. SMOTE-TL technique showed the improvement of 14%, 12%, 55%, 47%, 20% and 11% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively. SMOTE technique showed the improvement of 27%, 27%, 49%, 64%, 32% and 26% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively.

Table 6.5 shows the BL values of the models after developed on datasets balanced through SMOTE-TL and SMOTE-ENN. SMOTE-TL technique showed the improvement of 14%, 13%, 56%, 45%, 19% and 11% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively. SMOTE-ENN technique showed the improvement of 25%, 27%, 46%, 66%, 32% and 26% over Ant, Camel, Ivy, Jedit, Log4j and Prop imbalanced datasets respectively.

In case of extremely unbalanced dataset Jedit where there are only 2% of defective classes, oversampling techniques SMOTE-TL and SMOTE-ENN showed improvement of AUC values of 81% and 104% respectively.

It is evident that there is noteworthy increase in performance of models constructed after balancing the datasets through SMOTE-TL and SMOTE-ENN, where the predictive capability of later is very promising.

Table 6.5: Balance(BL) Results of SDP Models using data resampling techniques

Technique	Ant	Camel	Ivy	Jedit	Log4j	Prop
IMBALANCED	53.52	53.75	49.48	52.62	54.26	61.48

Table 6.5 continued from previous page

Technique	Ant	Camel	Ivy	Jedit	Log4j	Prop
ROS	61.2	61.81	77.34	78.22	61.92	64.7
SMOTE	58.95	57.86	75.79	83.31	62.7	64.71
BL-SMOTE	62.51	65.85	**78.12**	78.16	68.09	66.74
ADASYN	59.01	57.3	68.34	72.62	58.87	63.61
RUS	58.4	50.99	66.04	53.64	50	49.99
CC	51.11	59.79	46.94	53.64	56.27	59.96
NM	54.18	58.46	47.5	53.2	55.7	50.38
CNN	55.97	50.35	49.01	53.71	56.29	50.82
TL	56.3	54.79	50.78	59.13	49.62	58.83
ENN	63.44	48.69	28.71	51.99	59.39	42.32
SMOTE-TL	61.09	60.47	77.3	76.47	64.57	68.12
SMOTE-ENN	**67.08**	**68.49**	72.37	**87.5**	**71.69**	**77.22**

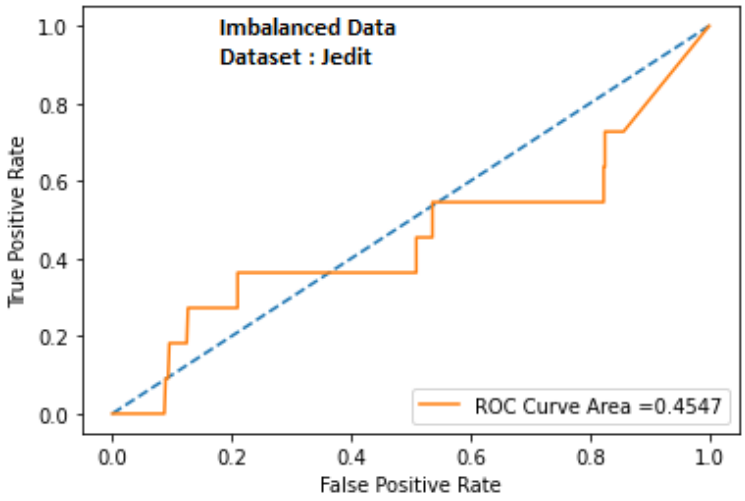

Figure 6.2: ROC of No Sampling on Jedit dataset

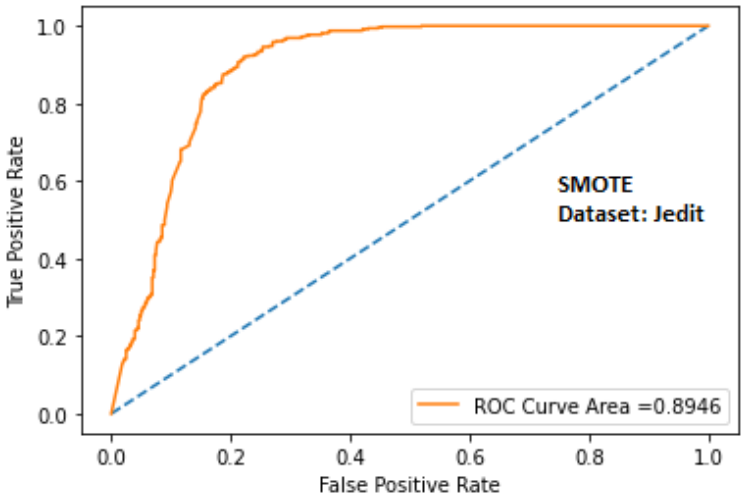

Figure 6.3: ROC of SMOTE on Jedit dataset

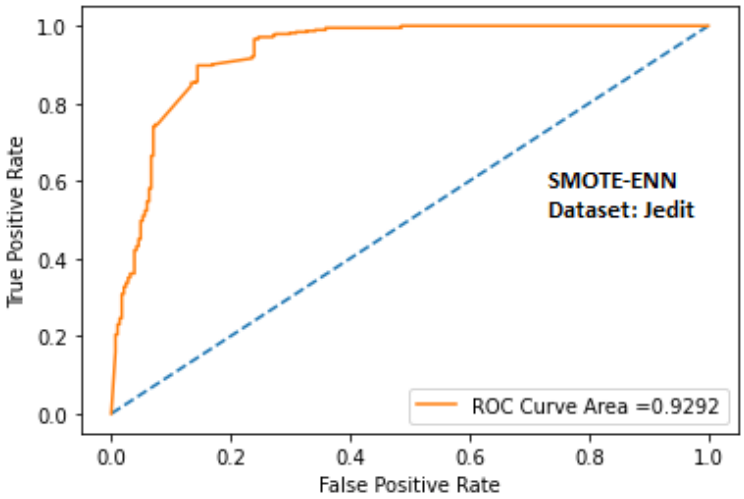

Figure 6.4: ROC of SMOTE-ENN Technique on Jedit dataset

RQ4. Which data resampling technique outperform the other techniques in ANN based defect prediction models.?

The study demonstrated a noteworthy improvement in the performance of SDP models, as measured by AUC, GM, and BL metrics. One of the aim of the research is to determine the most effective data resampling method for improving the predictive capability of SDP model. Statistical analysis is conducted with the Friedman test. This test evaluated AUC, GM, and BL values of SDP models constructed with different data resampling techniques examined in this work. The significance level was chosen at α=0.05 (confidence level of 95%), with 13 degrees of freedom (12 resampling methods and 1 without resampling).

Null hypothesis-H_{01}: The use of various data resampling methods (ROS, SMOTE, BL-SMOTE, ADASYN, RUS, CC, NM, CNN, TL, ENN, SMOTE-TL, and SMOTE-ENN) to balance imbalanced datasets did not yield a significant improvement in performance of SDP models measured by AUC.

Alternate hypothesis-H_{a1}: The AUC values of SDP models constructed after balancing the imbalanced datasets through resampling methods (ROS, SMOTE, BL-SMOTE, ADASYN, RUS, CC, NM, CNN, TL, ENN, SMOTE-TL, and SMOTE-ENN) have shown a significant improvement.

Null hypothesis-H_{02}: The use of various data resampling methods (ROS, SMOTE, BL-SMOTE, ADASYN, RUS, CC, NM, CNN, TL, ENN, SMOTE-TL, and SMOTE-ENN) to balance imbalanced datasets did not yield a significant improvement in performance of SDP models measured by GM.

Alternate hypothesis-H_{a2}: The GM values of SDP models constructed after balancing the imbalanced datasets through resampling methods (ROS, SMOTE, BL-SMOTE, ADASYN, RUS, CC, NM, CNN, TL, ENN, SMOTE-TL, and SMOTE-ENN) have

shown a significant improvement.

Null hypothesis-H_{03}: The use of various data resampling methods (ROS, SMOTE, BL-SMOTE, ADASYN, RUS, CC, NM, CNN, TL, ENN, SMOTE-TL, and SMOTE-ENN) to balance imbalanced datasets did not yield a significant improvement in performance of SDP models measured by BL.

Alternate hypothesis-H_{a3}: The BL values of SDP models constructed after balancing the imbalanced datasets through resampling methods (ROS, SMOTE, BL-SMOTE, ADASYN, RUS, CC, NM, CNN, TL, ENN, SMOTE-TL, and SMOTE-ENN) have shown a significant improvement.

Table 6.6 presents the outcomes of Friedman test performed to evaluate the performance of SDP models measured by AUC, GM, and BL. The obtained p-value was less than 0.05, which prove the outcomes are significant for all three performance metrics. And hence, we reject the null hypotheses H_{01}, H_{02}, and H_{03}. The mean ranks the resampling methods in terms of AUC, GM, and BL are also provided in Table 6.6. SMOTE-ENN achieved top rank in all three performance metrics, indicating its superior performance compared to other methods. Additionally, BL-SMOTE, SMOTE-TL, ROS, and SMOTE were ranked among the top five methods.

The findings also reveal that the performance of SDP models is significantly poor when datasets are imbalanced. Furthermore, ENN and NM were found to perform poorly in terms of AUC. The outcomes of Friedman test show SMOTE-ENN was the best data resampling method. To confirm this finding, we carried out post-hoc analysis through the Wilcoxon-signed rank test. The analysis compared the performance of models (AUC, GM, and BL) developed after SMOTE-ENN with

that of other methods.

The null and alternative hypothesis for the Wilcoxon-signed rank test in terms of AUC in this study are presented as follows:

H_{04}: $AUC_{SMOTE\text{-}ENN} = AUC_X$

H_{a4}: $AUC_{SMOTE\text{-}ENN} \neq AUC_X$

where X denotes ROS, SMOTE, BL-SMOTE, ADASYN, RUS, CC, NM, CNN, TL ENN and SMOTE-TL.

Similarly, the null and alternative hypothesis for the Wilcoxon-signed rank test in terms of GM are given as follows:

H_{05}: $GM_{SMOTE\text{-}ENN} = GM_X$

H_{a5}: $GM_{SMOTE\text{-}ENN} \neq GM_X$

where X denotes ROS, SMOTE, BL-SMOTE, ADASYN, RUS, CC, NM, CNN, TL ENN and SMOTE-TL.

Similarly, for the Wilcoxon-signed rank test in terms of BL are given as follows:

H_{06}: $BL_{SMOTE\text{-}ENN} = BL_X$

H_{a6}: $BL_{SMOTE\text{-}ENN} \neq BL_X$

where X denotes ROS, SMOTE, BL-SMOTE, ADASYN, RUS, CC, NM, CNN, TL ENN and SMOTE-TL.

Table 6.6: Results of Friedman Test

Technique	Mean Rank with respect to AUC	Mean Rank with respect to GM	Mean Rank with respect to BL
SMOTE-ENN	1.17 (Rank 1)	1.67 (Rank 1)	1.67 (Rank 1)
BL-SMOTE	2.33 (Rank 2)	2.67 (Rank 2)	2.50 (Rank 2)
SMOTE-TL	3.67 (Rank 3)	3.50 (Rank 3)	3.67 (Rank 3)
ROS	4.00 (Rank 4)	3.67 (Rank 4)	3.67 (Rank 3)
SMOTE	4.50 (Rank 5)	4.67 (Rank 5)	4.67 (Rank 4)
ADASYN	6.50 (Rank 6)	6.50 (Rank 6)	6.50 (Rank 5)
TL	8.83 (Rank 7)	9.17 (Rank 7)	9.17 (Rank 6)
RUS	9.33 (Rank 8)	10.00 (Rank 9)	9.83 (Rank 9)
CC	9.67 (Rank 9)	9.50 (Rank 8)	9.50 (Rank 7)
CNN	9.83 (Rank 10)	10.00 (Rank 9)	9.67 (Rank 8)
IMBALANCED	10.00 (Rank 11)	10.17 (Rank 10)	10.17 (Rank 11)
ENN	10.50 (Rank 12)	10.00 (Rank 9)	10.00 (Rank 10)
NM	10.67 (Rank 13)	9.50 (Rank 8)	10.00 (Rank 10)

The study used a level of confidence α = 0.05 and Bonferroni correction to reject null hypotheses H_{04}, H_{05}, and H_{06}. We compared 13 pairs of resampling methods by Wilcoxon test, the null hypotheses would be rejected if p-value obtained is greater than 0.05. Table 6.7 presents outcomes of Wilcoxon signed-rank test along with test statistics. The ”Sig” column in Table 6.7 shows significant difference in the AUC, GM and BL values of a pair of compared methods, while ”NotSig” denotes that there is no significant difference in the corresponding pair of methods. According to the test results, SMOTE-ENN is better than ROS, SMOTE, ADASYN, RUS, CC, NM, CNN, TL, ENN, SMOTE-TL, and NoResampling. However, BL-SMOTE was

not found to be significantly different (p-value $>$ 0.05 in terms of GM and BL) i.e., the performance of BL-SMOTE is comparable to that of SMOTE-ENN.

Table 6.7: Results of Wilcoxon Signed Rank Test

Resampling Techniques Pair	Test Statistics with respect to AUC	Test Statistics with respect to GM	Test Statistics with respect to BL
SMOTE-ENN vs IMBALANCED	S+ (p-value = 0.028)	S+ (p-value = 0.028)	S+ (p-value = 0.028)
SMOTE-ENN vs ROS	S+ (p-value = 0.028)	S+ (p-value = 0.046)	S+ (p-value = 0.046)
SMOTE-ENN vs SMOTE	S+ (p-value = 0.028)	S+ (p-value = 0.046)	S+ (p-value = 0.046)
SMOTE-ENN vs BL-SMOTE	S+ (p-value = 0.046)	S- (p-value = 0.116)	S- (p-value = 0.173)
SMOTE-ENN vs ADASYN	S+ (p-value = 0.028)	S+ (p-value = 0.028)	S+ (p-value = 0.028)
SMOTE-ENN vs RUS	S+ (p-value = 0.028)	S+ (p-value = 0.028)	S+ (p-value = 0.028)
SMOTE-ENN vs CC	S+ (p-value = 0.028)	S+ (p-value = 0.028)	S+ (p-value = 0.028)
SMOTE-ENN vs NM	S+ (p-value = 0.028)	S+ (p-value = 0.028)	S+ (p-value = 0.028)
SMOTE-ENN vs CNN	S+ (p-value = 0.028)	S+ (p-value = 0.028)	S+ (p-value = 0.028)
SMOTE-ENN vs TL	S+ (p-value = 0.028)	S+ (p-value = 0.028)	S+ (p-value = 0.028)
SMOTE-ENN vs ENN	S+ (p-value = 0.028)	S+ (p-value = 0.028)	S+ (p-value = 0.028)
SMOTE-ENN vs SMOTE-TL	S+ (p-value = 0.028)	S+ (p-value = 0.046)	S+ (p-value = 0.046)

6.2.6 Discussion

The study developed the efficient ANN-based SDP models using the six imbalanced open-source software datasets. The study used four over sampling techniques (ROS, SMOTE, BL-SMOTE, ADASYN), six under sampling techniques (RUS, CC, NM, CNN, TL, ENN) and two techniques combination of over sampling and under sam-pling (SMOTE-TL and SMOTE-ENN) to handle the problem of imbalance data. The study also developed

models with imbalanced data of six datasets to evaluate the performance of data resampling methods. Thus, a total of 6 x 13 = 78 defect prediction models were developed in this study. The study utilized stratified ten-fold cross validation to validate the models and assessed their performance with three reliable and consistent performance metrics, namely AUC, G-Mean, and Balance. One of the primary contributions of the study is to recommend the adoption of data resampling techniques for handling the problem of imbalanced data, as their use can significantly enhance the performance of SDP models. Additionally, this study employed statistical analysis to bolster the findings of the research. The conclusions of this study are presented as follows:

- The utilization of four oversampling techniques and two combined techniques of oversampling and undersampling significantly enhanced the performance of ANN-based SDP models.
- The performance measures AUC, G-Mean and Balance indicated that SMOTE-ENN technique outperformed other techniques. In case of a highly imbalanced da-taset from Jedit system, SMOTE-ENN technique showed the improvement of 104% over the performance of no resampling approach in terms of AUC.
- SMOTE-ENN, BL-SMOTE, SMOTE-TL, ROS and SMOTE are among best five performing techniques in terms of AUC, G-mean and Balance. The performance of BL-SMOTE is comparable with that of SMOTE-ENN in terms of G-mean and Balance.
- Although undersampling approaches RUS, CNN, CC, ENN and NM improved performance to an extent, they are ranked as poor data resampling techniques.

- The models developed with no resampling approach ranked worst in terms of G-Mean and Balance, and also amongst bottom three in terms of AUC.

6.3 Application of a Weighted Loss Function

The aim of this paper is to introduce a Weighted Loss function for Neural Networks (WL-NN), a cost-sensitive learning methodology designed for the classification of imbalanced software datasets. The proposed WL-NN assigns a higher cost to Type-II errors than Type-I errors by giving more weight to the defective classes. This instructs the model to focus more on the minority defect-prone class, consequently reducing the Type-II error rate. To achieve this goal, a comprehensive study using a dataset comprising twenty-two software systems extracted from the AEEEM [53], JIRA [54], and PROMISE [52] repositories is conducted. The performance of the proposed WL-NN approach is evaluated using the Area Under the Curve (AUC) obtained from Receiver Operating Characteristics (ROC) analysis, a widely accepted performance metric in defect prediction research [199]. Four types of defect prediction models developed in this study are:

i. NN: Neural network based defect prediction model over imbalanced data.

ii. WL-NN: Neural network with weighted loss function based defect prediction model over imbalanced data.

iii. NN + SMOTE: Neural network-based defect prediction model over balanced data through Synthetic Minority Oversampling with Edited Nearest Neighbor (SMOTE-ENN) technique. SMOTE-ENN was selected as it has been identified as a superior resampling technique in various studies [185] [199].

iv. WL-NN + SMOTE: Neural network with weighted loss function based defect prediction model over balanced data through SMOTE-ENN.

To accomplish our objective, this paper presents a tripartite study that addresses the following Research Questions (RQs):

RQ1: How well do the proposed WL-NN based SDP models perform on imbalanced datasets.? How does the proposed WL-NN improve the performance of SDP models over the imbalanced datasets.?

RQ2: What is the predictive performance of proposed WL-NN over balanced datasets.? How does the proposed WL-NN improve the performance of software prediction models over the balanced datasets.?

RQ3: Which approach outperforms the other approaches in neural network based SDP models.?

By addressing these research questions and offering substantial contributions, this paper significantly advances the field of software defect prediction and provides practical guidance for handling class imbalance.

6.3.1 Proposed Weighted Loss Function to Neural Network (WL-NN)

Neural Networks (NNs) represent a powerful machine learning methodology inspired by the intricate mechanisms of the human brain. Comprising interconnected units known as neurons, NNs exhibit a layered architecture that enables complex information processing. Within a Neural Network, each neuron obtains inputs from the preceding layer, which are

then individually multiplied by their associated weights and subsequently summed. This weighted sum undergoes an activation function, leading to the generation of an output signal. This output signal is then transmitted to the subsequent layer, facilitating the flow of information and computation throughout the network. The output of a neuron in the i^{th} layer is given by the following equation:

$$\widehat{y}_i \; = \; \phi_i \; \left(\sum_{j \, = \, 1}^{n} \left[x_j \; \times \; w_{ij} \right] \; + \; b_i \right) \tag{6.1}$$

Here x_j is the j^{th} input, w_{ij} is the weight between the j^{th} input and the current neuron, n is the total number of inputs to the neuron from the preceding layer, b_i is the bias term of the neuron and ϕ_i is the activation function for the current neuron. For each j in the range $[1, \; n]$, the multiplication of x_j with w_{ij} is summed up. The bias term is then added to this summation. Finally, the activation function ϕ_i is employed to transform this result into the output $\widehat{y}_i$. The activation function within a Neural Network plays a pivotal role in capturing and effectively modeling complex non-linear relationships. When it comes to predicting software defects, the association between software metrics and component defect proneness is frequently intricate and characterized by non-linearity. Thus, the utilization of a Neural Network emerges as a suitable approach for accurate software defect prediction [66].

During the iterative process, the Neural Network undergoes the sequential processing of input data, with subsequent calculation of the loss or error, also referred to as the misclassification cost. The purpose of the loss function, denoted as L, lies in quantifying the disparity between the expected outcome (y) and the outcome predicted ($\widehat{y}$), with the primary objective being the minimization of this loss during training. Selecting an

appropriate loss function is a pivotal decision, as it profoundly impacts the training and performance of neural networks. The choice of a loss function is intrinsically tied to the specific task at hand. For binary classification tasks, such as software defect prediction, one commonly employed loss function is Binary Cross-Entropy (BCE), also known as Log Loss or Logistic Loss. This loss function calculates the log-likelihood of the true labels under the predicted probability distribution. It assigns a higher loss for incorrect predictions and a lower loss for correct ones. By minimizing this loss function during model training, the model aims to improve its ability to predict the correct class labels, effectively learning to estimate probabilities for binary outcomes. Binary Cross-Entropy is particularly effective for tasks where the goal is to model probabilities, such as in binary classification problems and logistic regression. It can be defined by the following equation:

$$L_i = \frac{1}{n}\sum_{j=1}^{n}[-y_j \ \log \ \widehat{y_j} - (1-y_j)\log(1- \ \widehat{y_j})] \tag{6.2}$$

BCE loss is a convex function, which means that it has a unique minimum. This makes it easy to train machine learning models using BCE loss. BCE loss has several advantages over other loss functions such as mean squared error (MSE) loss. MSE tends to penalize larger errors heavily, which can be counterproductive in cases of imbalanced datasets where the defective class is in the minority. Binary Cross-Entropy, on the other hand, is better at handling imbalanced data as it focuses on the logarithm of predicted probabilities, effectively reducing the impact of outliers. Research studies have consistently demonstrated its efficacy in handling imbalanced data [200], making it a pragmatic choice for software defect prediction tasks.

In this study, defective class refers to positive class (class value: 1) and non-defective

class refers to negative class (class value: 0). The two types of errors that show up during binary classification are:

- Type-I Error: Misclassification of a negative (0) class as a positive (1) class.
- Type-II Error: Misclassification of a positive (1) class as a negative (0) class.

The algorithm-level method employed in the study introduces a weighted loss function as a solution to address the inherent challenges posed by imbalanced data. The conventional lost function assumes errors made with respect to different possible outcomes as same. The proposed classifier implements a weighted lost function which is given by the following equation:

$$\overline{L} = \frac{\sum_1^n \lambda_i \times L_i}{\sum_1^n \lambda_i} \tag{6.3}$$

where λ_i : weight of the i^{th} data point, L_i : loss of the i^{th} data point, and n : number of data points.

In the above equation, the weight-updated loss function is calculated by dividing the sum of the products of the loss of each data point and its corresponding weight, by the sum of the weights associated with each data point (n represents the total number of training data points). The proposed weights to loss function is given as the following:

$$\lambda_i = \begin{cases} 1 \quad for\ negative\ data\ point \\ \frac{n_p}{n_n}\ for\ positive\ data\ point \end{cases} \tag{6.4}$$

where n_p:number of positive data points and n_n:number of negative data points.

This error calculated with weighted loss function is then propagated back through the network and the weights are adjusted to minimize the error. This procedure is repeated till a stopping condition such as maximum iterations or minimum loss value is obtained.

The utilization of these weights in the loss function adjustment accounts for the discrepancies in class distribution, with a higher weight given to the positive data points (minority class), effectively addressing the data imbalance problem. Through this mechanism, the weighted loss function contributes significantly to the handling of data imbalance by enabling the neural network to learn more effectively from the minority class while maintaining accuracy on the majority class. The weights in this context are crucial as they represent the algorithm-level approach to enhance the model's ability to the imbalanced nature of the data.

6.3.2 Architecture of Proposed Neural Network

The proposed WL-NN architecture is presented in Table 6.8, outlining its structural configuration. Table 6.9 presents the values of various parameters consider in constructing the proposed neural network. The architecture and model parameters were determined heuristically. In table 1, the 'X' value in 'Shape' column represents 'batch size', while the second entry within the tuple denotes the number of neurons. The architecture and model parameters were established through a systematic trial and error process, considering the specific requirements of the study. As we incorporate varying batch sizes, the 'X' value within the architecture is not fixed and can vary accordingly. To implement the WL-NN, the Keras Framework in Python was employed. The subsequent sections present a detailed description of the individual components comprising WL-ANN, elucidating

their functionality and role within the framework.

Table 6.8: Structure of WL-NN

Layer	Layer Type	Shape	Activation Function
0	Input	(X, 20)	Relu
1	Dense	(X, 15)	Relu
2	Dense	(X, 10)	Relu
3	Dense	(X, 5)	Relu
4	Dense	(X, 1)	Sigmoid

Table 6.9: Parameter Settings of WL-NN

Parameter	Value
Regularizer	L2 regularization penalty of 0.001.
Epochs	10
Optimizer	Adam
Metrics	AUC
Loss function	Binary Cross Entropy

The proposed WL-NN used Rectified Linear Units (ReLU) as activation functions for the hidden(or intermediate) layers and Sigmoid activation function for the output layer.

ReLU Activation Function: The ReLU function presents itself as a horizontal line f(x)=0, when the unit is inactive i.e., $x<0$, and a lined inclined with slope 1, f(x) = x when the unit is active, i.e. x=0. The ReLU function sets all negative values to zeros.

Sigmoid Activation Function: A type of activation function also termed as a *squashing* function as it squeezes output into a range [0,1] which signifies the probability of classifying a component as 'defective'. Mathematically, this function has a characteristic 'S' shape, as

shown in Fig.2. The function is defined as:

$$f(x) \ = \ \frac{1}{1+e^{-x}} \tag{6.5}$$

Adam Optimization Algorithm: Stochastic gradient descent utilizes a fixed learning rate, commonly referred to as alpha, which remains constant for all weight updates throughout the training process. Adam optimization is a stochastic gradient descent method that computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients [201]. According to Kingma & Ba [201], the method is "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters."

Epochs: Epochs are basically the number of times the model runs its course through the dataset. So, one epoch is one cycle through the entire training dataset which includes the forward pass of the inputs through the network and a backward propagation of the error to adjust weights.

Batch sizes: Batch size refers to the number of data points that pass through the network in one iteration. Many such iterations form one epoch. Essentially, the number of iterations in one epoch can be given by the following equation:

$$i \ = \frac{N}{B} \tag{6.6}$$

where i is the number of iterations, N is the total number of data points and B is the batch size.

Training the neural network multiple times over increasing batch sizes is equivalent to decreasing learning rate of model, and is applicable to stochastic gradient descent. This

method was also found to have a shorter training time and have fewer parameter updates than actually decreasing the learning rate of the algorithm. Following this precedent, we trained our model four times with batch sizes increasing from x to y. The number of epochs was set to 10.

To compare performance of different approaches, statistical tests are conducted. We first used the Friedman test, a non-parametric test used to com-pare multiple sets of data for significant differences. This test is suitable when same parameter is measured under different conditions on same subject. In this study, the null hypothesis was defined as the defect prediction models constructed using different approaches exhibiting equal performance in terms of AUC. The Friedman test computed the rank of each approach across multiple datasets, and the average rank was determined as the mean rank for that specific technique. We then performed the Wilcoxon signed-rank test for post-hoc analysis. It is a statistical test that is used compare two related samples or repeated measurements on a single sample. In this study, we evaluate pair of techniques to examine the null hypothesis that there is no significant difference in the performance (AUC) of those techniques. The hypothesis testing of both Friedman test and Wilcoxon test are applied for a 95% confidence interval, corresponding to a significance level (α) of 0.05. These tests provided valuable insights into any significant differences that may exist between the approaches, ensuring a robust analysis and enabling informed conclusions to be drawn from the results.

6.3.3 Implementation

The implementation of the proposed approaches in the study is done in the Python programming language. The study utilized Keras, a deep learning API for Python [146], to construct the neural network architecture. The structure of the neural network (NN) is outlined in Table 6.8. The parameter settings of the NN is detailed in Table 6.9, while the remaining parameters are to their default values. The study employed the Imbalanced-learn API [197], which facilitated the implementation of various SMOTE-ENN technique. Additionally, scikit-learn [61], a comprehensive machine learning API, is utilized for data preprocessing, model fitting, model evaluation, and other related utilities. The statistical tests are conducted using IBM SPSS software [145].

6.3.4 Results

The study's research questions are addressed in this section by evaluating the performance of SDP models of various approaches. Table 6.10, Table 6.11 and Table 6.12 presents the AUC values of different approaches over AEEEM, JIRA and PROMISE repositories respectively. These tables provide comprehensive insights into the performance evaluation of the different approaches employed in the study and comparing their effectiveness.

RQ1. How well the proposed WL-NN based SDP models perform on imbalanced datasets.? How does the proposed WL-NN improve the performance of SDP models over the imbalanced datasets.?

To answer this RQ; we developed neural network based defect prediction models with proposed weighted loss function and analyzed the predictive capability of

WL-NN over twenty-two open source datasets of AEEEM, JIRA and PROMISE repositories. The performance of developed models is assessed using Area under ROC curve (AUC) (shown in Table 6.10, Table 6.11 and Table 6.12). The AUC values of NNWL over imbalanced datasets of EQ, JDT, PDE, Lucene and Mylyn systems from AEEEM repository are 0.8245, 0.8433, 0.7833, 0.7920 and 0.7564 respectively i.e., there is an improvement in performance of 5.75%, 3.49%, 3.39%, 4.28% and 4.14% respectively. The AUC values of WL-NN over imbalanced datasets of Active MQ, Derby, Groovy, Hbase, Hive, JRuby and Wicket systems from JIRA repository are 0.8741, 0.7991, 0.8441, 0.8375, 0.8259, 0.8912 and 0.7838 respectively i.e., there is an improvement in performance of 2.05%, -0.03%, 1.93%, 11.92%, 2.70%, 1.53% and -1.33% respectively. The AUC values of WL-NN over imbalanced datasets of Ant, Camel, Ivy, Jedit, Log4j, Poi, Tomcat, Velocity, Xalan and Xerces systems from PROMISE repository are 0.8073, 0.6783, 0.7081, 0.7653, 0.5718, 0.7448, 0.7544, 0.6957, 0.7452 and 0.8099 respectively i.e., there is an improvement in performance of 0.99%, 4.31%, 38.90%, 17.18%, -16.99%, 30.99%, -1.69%%, 17.42%, 4.43% and 15.03% respectively. Eighteen out of the twenty-two datasets have shown improvement in AUC values with application of the weighted loss function in neural network. Thus the predictive capability of the proposed model is found to be significant.

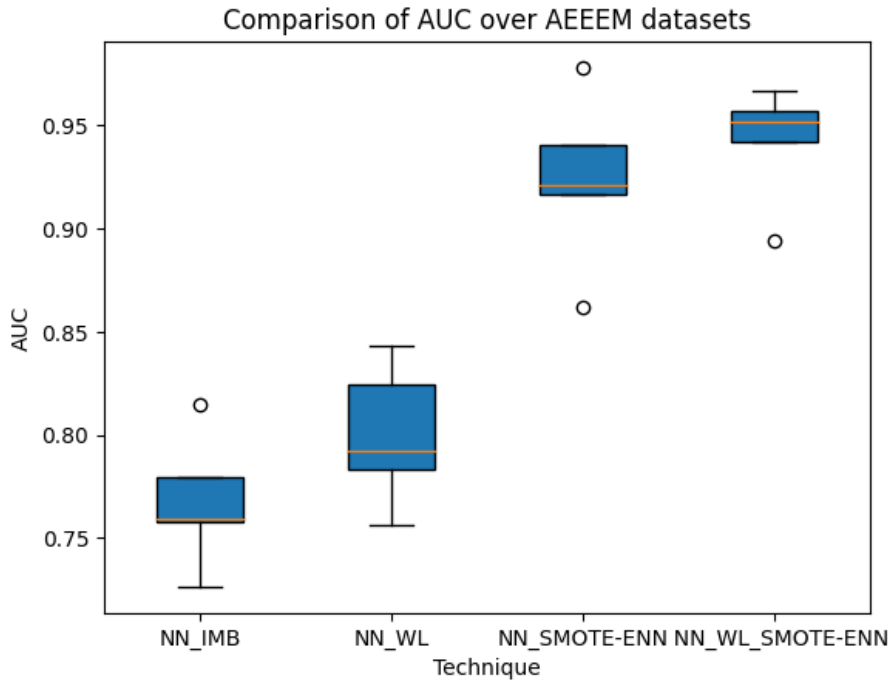

Figure 6.5: Box plot of AUC values of different models over AEEEM datasets

Table 6.10: AUC values of proposed models over datasets of AEEEM repository

Approach / Dataset	**NN over Imbalanced datasets**	**WL-NN over Imbalanced datasets**	**NN over balanced datasets (NN + SMOTE-ENN)**	**WL-NN over balanced datasets (WL-NN + SMOTE-ENN)**
EQ	0.7797	0.8245	**0.9781**	0.9574
JDT	0.8149	0.8433	0.9402	**0.967**
PDE	0.7576	0.7833	0.9163	**0.9518**
Lucene	0.7595	0.792	0.9209	**0.9419**
Mylyn	0.7263	0.7564	0.8622	**0.8939**

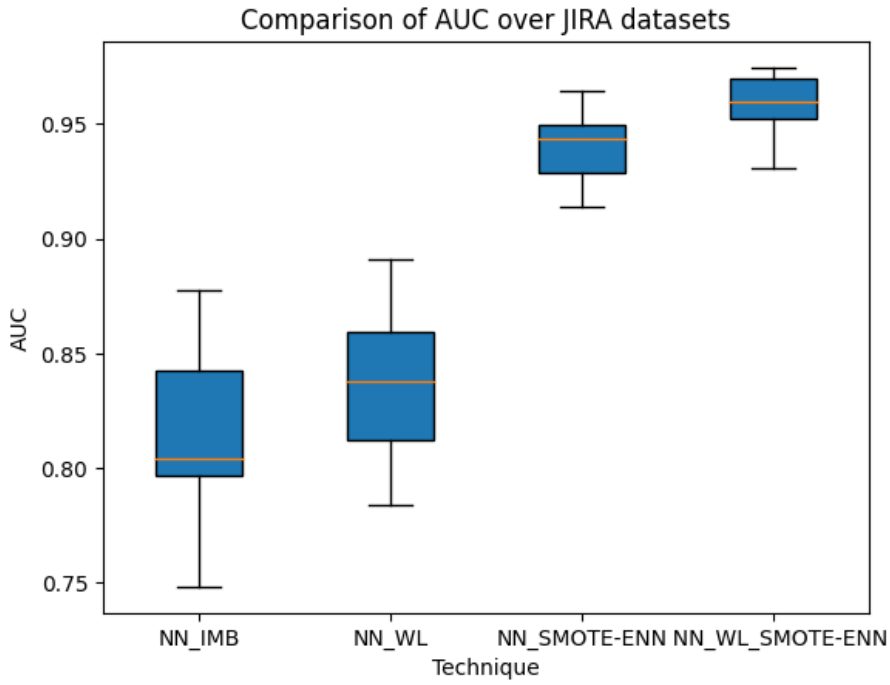

Figure 6.6: Box plot of AUC values of different models over JIRA datasets

Table 6.11: AUC values of proposed models over datasets of JIRA repository

Approach / Dataset	**NN over Imbalanced datasets**	**WL-NN over Imbalanced datasets**	**NN over balanced datasets (NN + SMOTE-ENN)**	**WL-NN over balanced datasets (WL-NN + SMOTE-ENN)**
Active MQ	0.8565	0.8741	**0.9643**	**0.9745**
Derby	0.7993	0.7991	0.914	**0.9305**
Groovy	0.8281	0.8441	0.9452	**0.9594**
Hbase	0.7483	0.8375	0.9434	**0.957**

Table 6.11 continued from previous page

Approach / Dataset	NN over Imbalanced datasets	WL-NN over Imbalanced datasets	NN over balanced datasets (NN + SMOTE-ENN)	WL-NN over balanced datasets (WL-NN + SMOTE-ENN)
Hive	0.8042	0.8259	0.923	**0.9477**
JRuby	0.8778	0.8912	0.9545	**0.9746**
Wicket	0.7944	0.7838	0.935	**0.9649**

Table 6.12: AUC values of proposed models over datasets of PROMISE repository

Approach / Dataset	NN over Imbalanced datasets	WL-NN over Imbalanced datasets	NN over balanced datasets (NN + SMOTE-ENN)	WL-NN over balanced datasets (WL-NN + SMOTE-ENN)
Ant	0.7994	0.8073	**0.9295**	**0.9245**
Camel	0.6503	0.6783	0.8068	**0.8079**
Ivy	0.5098	0.7081	0.8628	**0.9231**
Jedit	0.6531	0.7653	0.8576	**0.9064**
Log4j	0.6888	0.5718	0.8077	**0.8397**
Poi	0.5686	0.7448	**0.8666**	**0.8481**
Tomcat	0.7674	0.7544	0.9136	**0.9454**
Velocity	0.5925	0.6957	0.7625	**0.9006**
Xalan	0.7136	0.7452	0.856	**0.9089**
Xerces	0.7041	0.8099	0.9112	**0.9174**

RQ2. What is the predictive performance of proposed WL-NN over balanced datasets

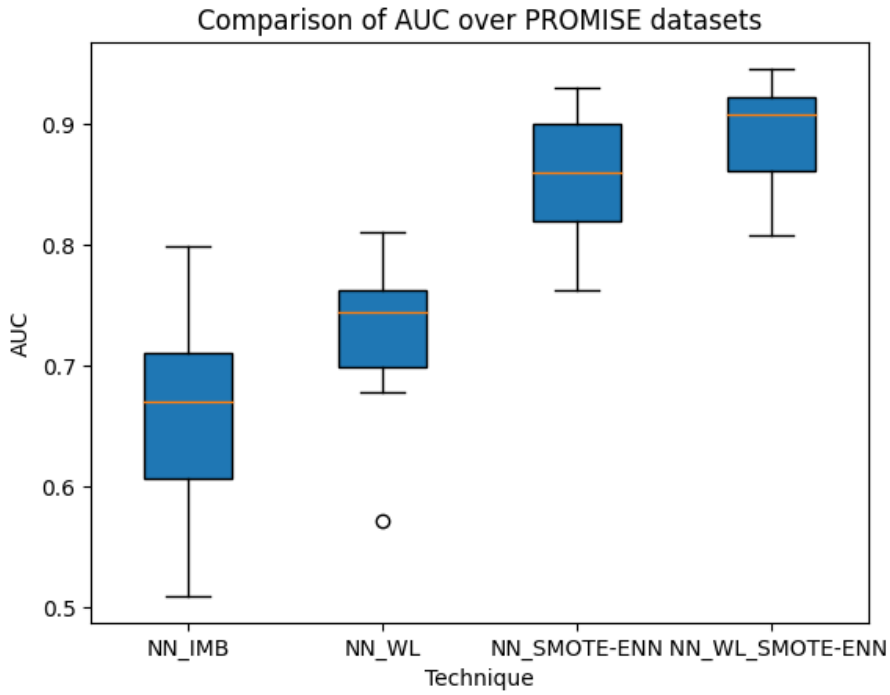

Figure 6.7: Box plot of AUC values of different models over PROMISE datasets

(WL-NN + SMOTE-ENN).? How does the proposed WL-NN improve the performance of software prediction models over the balanced datasets.?

To answer this RQ; we first applied SMOTE-ENN data sampling technique over the over twenty-two imbalanced datasets of AEEEM, JIRA and PROMISE repositories. Then the proposed weighted loss neural network based defect prediction models are developed over the balanced datasets and analyzed the predictive capability of WL-NN on balanced datasets. The performance of the developed models is assessed using Area under ROC curve (AUC) (shown in Table 6.10, Table 6.11 and Table 6.12). The AUC values of WL-NN over balanced datasets of EQ, JDT, PDE, Lucene

and Mylyn systems from AEEEM repository are 0.9574, 0.9670, 0.9518, 0.9419 and 0.8939 respectively i.e., there is an improvement in performance of 22.79%, 18.66%, 25.63%, 24.02% and 23.08% respectively. The AUC values of NNWL over imbalanced datasets of Active MQ, Derby, Groovy, Hbase, Hive, JRuby and Wicket systems from JIRA repository are 0.9745, 0.9305, 0.9594, 0.957, 0.9477, 0.9746 and 0.9649 respectively i.e., there is an improvement in performance of 13.78%, 16.41%, 15.86%, 27.89%, 17.84%, 11.03% and 21.46% respectively. The AUC values of NNWL over imbalanced datasets of Ant, Camel, Ivy, Jedit, Log4j, Poi, Tomcat, Velocity, Xalan and Xerces systems from PROMISE repository are 0.9245, 0.8079, 0.9231, 0.9064, 0.8397, 0.8481, 0.9454, 0.9006, 0.9089 and 0.9174 respectively i.e., there is an improvement in performance of 15.65%, 24.23%, 81.07%, 38.78%, 21.91%, 49.16%, 23.20%, 52%, 27.37% and 30.29% respectively. All the datasets have shown significant improvement in AUC values with the application of both SMOTE-ENN and weighted loss function in neural network. Thus the predictive capability of the proposed model is found to be very significant.

RQ3. Which approach outperform the other approaches in neural network based SDP models.?

The AUC performance measure is evaluated for the SDP models over all the datasets examined in this study. Figure 1.5, Figure 1.6, and Figure 1.7 show the box plots depicting the AUC values of the models developed in the study over AEEEM, JIRA and PROMISE repositories respectively. In order to address this research question, we conducted a comprehensive statistical analysis employing the Friedman test and Wilcoxon-signed rank test. The Friedman test was chosen as the statistical tool and

conducted with a significance level (α) of 0.05. The null hypothesis associated with the Friedman test was formulated and tested as follows:

> *Null hypothesis-H_{01}:* The defect prediction models developed with the proposed weighted loss function for neural networks did not yield a significant improvement in performance over the uniform weighted function.
>
> *Alternate hypothesis-H_{a1}:* The defect prediction models developed with the proposed weighted loss function for neural have shown significant improvement in the performance over the uniform weighted function.

Table 6.13 presents the outcomes of Friedman test conducted to evaluate the performance of SDP models. The obtained p-value was less than 0.05, which prove the outcomes are significant. And hence, we reject the null hypotheses H_{01}. The mean ranks of all the four approaches (NN, WL-NN, NN + SMOTE-ENN and WL-NN + SMOTE-ENN) are also provided in Table 10. Higher mean rank value in the table indicates the better approach. The results show that WL-NN + SMOTE-ENN obtained best rank over that NN + SMOTE-ENN and WL-NN obtained better rank over NN. It is to be noted that WL-NN + SMOTE-ENN approach has obtained the best rank of all the approaches. To confirm this finding, post-hoc analysis through the Wilcoxon-signed rank test is carried out. The analysis compared the performance of models developed with WL-NN + SMOTE-ENN approach with that of other approaches.

The null and alternative hypothesis for the Wilcoxon-signed rank test in terms of AUC in this study are presented as follows:

H_{02}: $\text{AUC}_{\text{WL-NN + SMOTE-ENN}} = \text{AUC}_{\text{X}}$

H_{a2}: $\text{AUC}_{\text{WL-NN + SMOTE-ENN}} \neq AUC_{\text{X}}$ where X denotes NN, WL-NN and SMOTE-ENN.

The study performed the pair-wise comparison of four approaches through Wilcoxon signed rank test, the null hypotheses would be rejected if p-value obtained is greater than 0.05. The study used a level of confidence α = 0.05 and Bonferroni correction to reject null hypotheses H_{02}. Table 6.14 presents outcomes of Wilcoxon signed-rank test along with test statistics. In Table 6.14, S+ denotes a significant difference in the performance of a pair of compared techniques, while S- indicates no significant difference in the corresponding pair of techniques. The results of the Wilcoxon signed-rank test indicate that WL-NN + SMOTE-ENN approach is significantly superior to other approaches (NN + SMOTE, WL-NN, and NN) except over AEEEM datasets. The proposed weighted loss function for neural networks shows significantly superior over uniform function approaches.

Table 6.13: Results of Friedman Test

Approach	**Mean Rank over AEEEM datasets**	**Mean Rank over JIRA datasets**	**Mean Rank over PROMISE datasets**
WL-NN + SMOTE-ENN	3.80 (Rank 1)	4.00 (Rank 1)	3.80 (Rank 1)
NN + SMOTE-ENN	3.20 (Rank 2)	3.00 (Rank 2)	3.20 (Rank 2)
WL-NN	2.00 (Rank 3)	1.71 (Rank 3)	1.80 (Rank 3)
NN	1.00 (Rank 4)	1.28 (Rank 4)	1.20 (Rank 4)

Table 6.14: Results of Wilcoxon Signed Rank Test

Approach	Test statistics over AEEEM datasets	Test statistics over JIRA datasets	Test statistics over PROMISE datasets
(WL-NN + SMOTE-ENN) vs (NN + SMOTE-ENN)	S- (p-value = 0.08)	S+ (p-value =0.01)	S+ (p-value =0.028)
(WL-NN + SMOTE-ENN) vs (WL-NN)	S+ (p-value = 0.04)	S+ (p-value =0.01)	S+ (p-value =0.005)
(WL-NN + SMOTE-ENN) vs (NN)	S+ (p-value = 0.04)	S+ (p-value =0.01)	S+ (p-value =0.005)
(NN + SMOTE-ENN) vs (WL-NN)	S+ (p-value = 0.04)	S+ (p-value =0.01)	S+ (p-value =0.005)
(NN + SMOTE-ENN) vs (NN)	S+ (p-value = 0.04)	S+ (p-value =0.01)	S+ (p-value =0.005)
(WL-NN) vs (NN)	S+ (p-value = 0.04)	S+ (p-value =0.02)	S+ (p-value =0.01)

6.3.5 Discussion

The study addressed the challenge of imbalanced data in software defect prediction (SDP) by proposing a Weighted Loss Function for Neural Networks (WL-NN). The four types of defect prediction models constructed in the study are: NN over imbalanced data, WL-NN over imbalanced data, NN over balanced data via SMOTE-ENN (NN + SMOTE-ENN), and WL-NN over balanced data via SMOTE-ENN (WL-NN + SMOTE-ENN). The experiments are conducted on twenty-two open source datasets sourced from AEEEM, JIRA, and PROMISE repositories. Thus, a total of 4 x 22 = 88 defect prediction models were developed in this study. The study utilized stratified ten-fold cross validation to validate the models and assessed their performance with three reliable and consistent performance metric, AUC. The results of our study clearly demonstrate the efficacy of

the proposed WL-NN approach in improving the performance of SDP models. The conclusions of the study are reported as follows:

- WL-NN alone has shown improvement of over 7% in performance of defect prediction models.
- WL-NN, when combined with SMOTE-ENN has shown improvement of over 27% in performance of defect prediction models.
- WL-NN + SMOTE-ENN outperformed all other approaches, exhibiting the highest predictive performance among the evaluated models. The order of performance of four types of defect prediction models are:

  WL-NN + SMOTE-ENN $>$ NN + SMOTE-ENN $>$ WL-NN $>$ NN

The findings of the study highlight the significance of incorporating a weighted loss function in neural networks, along with data resampling, to effectively address the challenges posed by imbalanced data in software defect prediction. The study also involved statistical analysis of the results produced in order to strengthen the conclusions of the study.

6.4 Discussion

This study aimed to address the challenge of imbalanced data in software defect prediction (SDP) using Artificial Neural Networks (ANN) by applying various data resampling techniques and proposing a Weighted Loss Function (WL-NN). A total of 78 ANN-based models were developed using 12 data resampling techniques across six open-source

datasets, while 88 models were created using WL-NN and its combinations across 22 datasets. Stratified ten-fold cross-validation and three performance metrics (AUC, G-Mean, and Balance) were employed for validation. The key conclusions are:

- **Data Resampling Techniques:** The use of oversampling, particularly SMOTE-ENN and hybrid methods, significantly improved the performance of ANN-based models. SMOTE-ENN was the top performer, enhancing model performance by up to 104% in highly imbalanced datasets.
- **Weighted Loss Function:** The introduction of WL-NN showed a 7% improvement in performance on its own, while combining WL-NN with SMOTE-ENN led to a 27% improvement, outperforming all other approaches.
- **Top Techniques:** The best-performing models were those combining WL-NN with SMOTE-ENN, followed by NN with SMOTE-ENN. The ranking of approaches was: WL-NN + SMOTE-ENN $>$ NN + SMOTE-ENN $>$ WL-NN $>$ NN.
- **Underperforming Techniques:** While undersampling techniques like RUS and CNN provided moderate improvements, they were less effective than oversampling and hybrid approaches. Models developed without resampling showed the poorest performance.

In conclusion, this study advocates the use of oversampling techniques and weighted loss functions in ANN for handling imbalanced data, as they substantially improve SDP model performance. Statistical analyses further support the robustness of these findings.

Chapter 7

Swarm Intelligence-Based Feature Selection for Software Defect Prediction

7.1 Introduction

In the ever-evolving landscape of software engineering, the quest for developing reliable and high-quality software systems remains a paramount objective. Among the myriad challenges that software developers and engineers face, perhaps one of the most pervasive is the presence of defects or bugs within software code [202]. These defects, if left aundetected or unresolved, can have far-reaching consequences, ranging from degraded system performance and compromised user experience to significant financial losses and reputational damage for organizations. Consequently, the task of predicting and mitigating software defects has emerged as a critical area of research and development within the field of software engineering.

Software defect prediction (SDP) aimed at identifying and mitigating potential defects in software systems before they manifest into critical issues during deployment or operation [203]. By leveraging historical data, code metrics, and various other software attributes, defect prediction models endeavor to identify patterns and trends that may indicate the presence of defects within software code. The ultimate goal of defect prediction is to enable software developers and quality assurance teams to proactively address and rectify potential defects, thereby enhancing the overall reliability, stability, and quality of software systems [204].

However, despite the advancements in software engineering practices and tools, the accurate prediction of software defects remains a daunting task. One of the primary challenges in defect prediction lies in the vast and often redundant set of features used to characterize software systems [205]. Without appropriate feature selection techniques, models may suffer from overfitting, dimensionality curse, and poor generalization, leading to suboptimal performance and limited interpretability. Feature selection, a fundamental preprocessing step in machine learning, aims to identify the most relevant and informative subset of features from a larger feature space [206]. By eliminating redundant or irrelevant features, feature selection enhances model efficiency, reduces computational complexity, and improves model interpretability. Various techniques for feature selection exist, including filter methods, wrapper methods, and embedded methods, each with its own strengths and limitations [206]. Filter methods, such as chi-square and information gain, assess the relevance of features independently of the learning algorithm, making them computationally efficient and suitable for high-dimensional datasets. Wrapper methods, on the other hand, evaluate feature subsets based on their predictive performance using a specific learning algorithm, making them more computationally intensive but potentially

more effective. Embedded methods integrate feature selection directly into the model training process, optimizing feature selection and model building simultaneously.

In recent years, researchers have explored innovative approaches to feature selection inspired by natural phenomena, such as swarm intelligence [207]. Swarm intelligence leverages the collective behavior of decentralized and self-organized systems to solve complex optimization problems [208]. Algorithms such as Ant Colony Optimization (ACO) [209], Cuckoo Search (CS) [210], and Crow Search (CRS) [211] mimic the foraging behavior of social organisms to efficiently explore and exploit search spaces, making them promising candidates for feature selection in software defect prediction. This study aims to investigate the efficacy of swarm intelligence techniques, specifically ACO, CS, and CRS, in comparison to traditional filter-based methods such as chi-square and information gain, for feature selection in software defect prediction. The following research questions are framed to guide the investigation of this study:

RQ1: What is the predictive capability of swarm intelligence techniques for feature selection in software defect prediction?

RQ2: How do swarm intelligence techniques compare to traditional filter-based methods in terms of feature selection performance for software defect prediction?

RQ3: How does the performance of machine learning models vary across different software projects and datasets when incorporating feature selection techniques?

RQ4: Which swarm intelligence technique demonstrates the highest efficacy in enhancing the performance of machine learning models for software defect prediction?

To address these research questions, the study conducted extensive experimentation on the defect datasets of twenty-two java systems from three repositories, namely - AEEEM [53], JIRA [54] and PROMISE [52]. The machine learning algorithms employed to build defect prediction models and assess the impact of feature selection techniques on their performance are Logistic Regression, Support Vector Machine, Naïve Bayes, and Random Forest.

The remainder of this chapter is organized as follows: Section 2 provides an overview of the swarm intelligence techniques employed in this study. The results are presented and analyzed in Section 3, followed by discussion of the study in Section 4.

7.2 Swarm Intelligence-Based Feature Selection Techniques

The study proposed the investigation of the following feature selection techniques for defect prediction:

Ant Colony Optimization (ACO)

ACO is a swarm intelligence-based optimization algorithm inspired by the foraging behavior of ants. In ACO, artificial ants traverse a solution space, depositing pheromone trails that guide subsequent iterations towards promising solutions. The formula for updating pheromone trails in ACO is given by:

$$\tau_{ij} = (1-\rho).\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{7.1}$$

Where τ_{ij} is the pheromone level on edge (*i,j*), ρ is the pheromone evaporation rate,

$\Delta\tau_{ij}^{k}$ is the amount of pheromone deposited by ant k on edge *(i,j)*, and *m* is the number of ants.

ACO is a robust and flexible optimization algorithm that can effectively explore complex solution spaces and identify optimal feature subsets. It is capable of handling high-dimensional data and nonlinear relationships between features. ACO may suffer from convergence issues and parameter sensitivity, requiring careful tuning of parameters such as pheromone evaporation rate and exploration-exploitation balance.

Cuckoo Search (CS)

CS is a metaheuristic optimization algorithm inspired by the brood parasitism of cuckoo birds. In CS, each cuckoo lays eggs in randomly chosen nests, and the nests with higher fitness values are retained for the next iteration. The formula for updating nest positions in CS is given by:

$$x_i^{t+1} = \ x_i^t + \alpha.\ L\ .\ Levy(\lambda) \tag{7.2}$$

Where x_i^t is the position of cuckoo *i* at iteration *t*, α is the step size, *L* is the learning rate, and Levy(λ) is the Levy flight distribution.

CS is a simple yet powerful optimization algorithm that exhibits fast convergence and good exploration-exploitation balance. It is suitable for high-dimensional optimization problems and can handle non-convex and multimodal search spaces. CS may suffer from premature convergence and parameter sensitivity, particularly with regard to the step size and learning rate parameters.

Crow Search (CRS)

CRS is a metaheuristic optimization algorithm inspired by the social foraging behavior of crows. In CRS, a population of crows searches for food sources in the solution space, with each crow adjusting its position based on the locations of neighboring crows. The formula for updating crow positions in CRS is given by:

$$x_i^{t+1} = \ x_i^t + \ \alpha. \sum_{j=1}^{N} w_j. \ (x_j - \ x_i) + \beta. \ Levy(\lambda) \tag{7.3}$$

Where x_i^t is the position of crow *i* at iteration *t*, α and β are the step sizes, w_j is the weight assigned to crow *j*, x_j is the position of crow *j*, and Levy(λ) is the Levy flight distribution.

CRS is a scalable and efficient optimization algorithm that combines global exploration with local exploitation. It is capable of handling multimodal and non-convex search spaces and can effectively identify diverse solutions. CRS may require fine-tuning of parameters such as step sizes and weights to achieve optimal performance. It may also suffer from slow convergence in certain scenarios.

7.3 Results

The results of this study are presented in terms of AUC values in Table 7.1, Table 7.2 and Table 7.3 for each classifier-feature selection technique across the datasets from the AEEEM, JIRA, and PROMISE repositories respectively. The AUC values provide a measure of the classification accuracy for predicting defective software modules, with

higher values indicating better performance.

Table 7.1: AUC Values of Feature Selection Techniques over Datasets of PROMISE Repository

Representation of Approach	ML Technique	FS Technique	Ant	Camel	Ivy	jEdit	Log4j	Lucene	pBeans	POI	Prop	Synapse
LR_None	**LR**	**None**	0.5950	**0.6889**	0.7143	**0.5646**	0.5731	0.7586	0.4500	0.8170	0.7170	0.7459
LR_CHI2		**CHI2**	0.6321	0.6656	0.7540	0.4411	0.6469	0.7530	0.5500	0.8154	0.6690	0.7433
LR_IG		**IG**	0.6142	0.6637	0.7628	**0.5791**	0.6620	0.7686	0.6500	0.8165	0.6991	0.7145
LR_ACO		**ACO**	**0.6309**	0.6769	**0.7497**	**0.7000**	0.6753	0.7364	**0.7500**	0.8214	0.7429	**0.7629**
LR_CS		**CS**	0.6231	**0.6828**	**0.7651**	0.6936	**0.7120**	**0.7713**	0.5667	**0.8361**	**0.749**	0.7411
LR_CSA		**CSA**	**0.6355**	**0.6893**	**0.7446**	0.6625	**0.6687**	0.7884	0.6000	0.8257	0.7431	0.7766
SVM_None	**SVM**	**None**	0.5645	0.6248	0.6063	0.2981	0.3892	0.6826	0.3500	0.7939	0.3300	0.6803
SVM_CHI2		**CHI2**	**0.5770**	0.6018	0.6183	0.3575	0.4154	0.6830	0.5000	0.7937	0.4766	0.6824
SVM_IG		**IG**	0.5665	0.5486	0.5975	0.2732	0.5105	0.6358	0.4750	0.7937	0.4546	0.6915
SVM_ACO		**ACO**	**0.5529**	0.5366	**0.6678**	**0.2825**	0.4363	0.7518	**0.7750**	**0.8343**	0.4738	0.7093
SVM_CS		**CS**	0.4173	**0.5443**	**0.6145**	**0.2925**	**0.6558**	**0.7625**	0.7500	0.8169	0.5405	**0.7369**
SVM_CSA		**CSA**	**0.5320**	**0.6296**	0.6353	**0.5228**	**0.5839**	0.7469	0.5500	0.8285	**0.5614**	0.7350
NB_None	**NB**	**None**	0.6170	0.6668	0.7431	0.6274	0.6362	0.7267	0.4750	0.7901	0.6919	0.7326
NB_CHI2		**CHI2**	0.6396	0.6500	0.7899	0.6200	0.6849	0.6876	**0.7000**	0.7931	0.7075	0.7100
NB_IG		**IG**	**0.6206**	0.6406	0.8048	**0.7126**	0.6659	0.7267	**0.7000**	0.7781	0.6918	**0.7717**
NB_ACO		**ACO**	0.6310	**0.6717**	**0.8084**	**0.7378**	0.6487	0.7386	**0.7000**	0.8024	0.7036	0.7273
NB_CS		**CS**	**0.6464**	0.6709	**0.7306**	0.6534	**0.7322**	**0.7442**	**0.7000**	0.7957	**0.7229**	0.7308
NB_CSA		**CSA**	**0.6529**	**0.671**	0.8078	0.6339	**0.6504**	0.7267	0.5000	**0.8033**	0.7049	0.7440
RF_None	**RF**	**None**	**0.645**	**0.6962**	0.7712	0.8087	**0.7043**	0.8088	0.7167	0.8868	0.7399	0.8403
RF_CHI2		**CHI2**	0.6127	0.6594	**0.8026**	0.7248	0.6817	0.8016	0.7375	0.8847	0.7334	0.8281
RF_IG		**IG**	**0.6377**	0.6842	**0.7652**	0.753	**0.7848**	0.7908	**0.7708**	0.8963	0.7455	**0.8515**
RF_ACO		**ACO**	0.6302	**0.6757**	0.7705	**0.7774**	0.7893	0.8009	0.7667	0.8894	0.7662	0.7695
RF_CS		**CS**	0.6439	0.6811	**0.7344**	**0.7697**	0.7003	0.7783	0.6667	0.8932	**0.7915**	0.8339
RF_CSA		**CSA**	0.6348	0.6912	0.7718	**0.8253**	**0.7841**	**0.8029**	0.7125	**0.8897**	0.7717	0.8328

Table 7.2: AUC Values of Feature Selection Techniques over Datasets of AEEEM Repository

Representation of Approach	ML Technique	FS Technique	EQ	JDT	Lucene	Mylyn	PDE
LR_None	**LR**	**None**	0.7795	**0.8415**	0.7418	**0.7767**	0.7202
LR_CHI2		**CHI2**	0.8218	0.8059	0.7522	0.7606	0.732
LR_IG		**IG**	0.8238	0.8257	0.81	**0.709**	0.7487
LR_ACO		**ACO**	**0.8263**	0.8396	**0.796**	0.7766	0.7153
LR_CS		**CS**	0.818	**0.8581**	0.7886	0.7519	**0.7418**
LR_CSA		**CSA**	**0.8171**	0.8378	**0.7964**	0.7391	**0.7494**
SVM_None	**SVM**	**None**	0.7874	0.7907	0.6949	0.6474	0.625
SVM_CHI2		**CHI2**	0.7803	0.792	0.6821	0.5947	0.6354
SVM_IG		**IG**	0.7849	0.7545	0.6453	0.6322	0.6351
SVM_ACO		**ACO**	**0.8053**	0.8499	**0.7177**	**0.7453**	0.71
SVM_CS		**CS**	0.7871	**0.8553**	**0.7338**	**0.7475**	**0.7036**
SVM_CSA		**CSA**	**0.7884**	**0.8511**	0.7064	0.704	**0.6772**
NB_None	**NB**	**None**	0.8154	0.8128	0.7769	0.7268	0.7603
NB_CHI2		**CHI2**	0.7866	0.8108	0.7244	0.733	0.7146
NB_IG		**IG**	**0.8094**	0.8221	0.7868	**0.7631**	0.755
NB_ACO		**ACO**	0.8189	**0.8187**	**0.7837**	0.7448	0.7482
NB_CS		**CS**	**0.8225**	0.8198	**0.7869**	0.7486	**0.7646**
NB_CSA		**CSA**	0.8113	**0.8224**	0.7841	0.7451	**0.7651**
RF_None	**RF**	**None**	**0.8557**	**0.8824**	0.8178	0.8258	**0.7905**
RF_CHI2		**CHI2**	0.8321	0.8434	0.7356	0.792	0.7235
RF_IG		**IG**	**0.8601**	0.8428	**0.7778**	0.8097	0.7589
RF_ACO		**ACO**	0.8462	**0.8789**	0.7974	**0.8143**	0.7803
RF_CS		**CS**	0.8494	0.8765	**0.8267**	**0.828**	0.79
RF_CSA		**CSA**	0.836	0.874	0.7909	0.826	**0.792**

Table 7.3: Values of Feature Selection Techniques over Datasets of JIRA Repository

Representation of Approach	ML Technique	FS Technique	Active MQ	Derby	Groovy	Hbase	Hive	JRuby	Wicket
LR_None	**LR**	**None**	0.854	**0.8185**	0.6854	0.6633	0.7325	0.6731	0.7974
LR_CHI2		**CHI2**	0.8026	0.3217	0.6043	0.4673	0.6854	0.4132	0.7914
LR_IG		**IG**	0.8703	0.7803	0.5317	**0.8572**	0.7955	**0.9568**	0.8111
LR_ACO		**ACO**	0.8682	0.8108	**0.7595**	0.8156	0.8218	0.9024	**0.8207**
LR_CS		**CS**	0.8664	0.8033	0.752	0.6444	**0.8229**	0.9161	0.7992
LR_CSA		**CSA**	**0.8715**	0.8015	0.7367	0.8245	0.8098	0.9102	0.8077
SVM_None	**SVM**	**None**	0.8237	0.6923	0.2859	0.4491	0.8116	**0.8924**	0.5407
SVM_CHI2		**CHI2**	0.7005	0.6799	0.2931	0.5043	0.8121	0.8882	0.7646
SVM_IG		**IG**	0.7084	0.6036	0.5079	0.7203	0.8588	0.7113	0.6261
SVM_ACO		**ACO**	0.8127	0.6283	**0.7963**	**0.7662**	0.8742	0.809	**0.8288**
SVM_CS		**CS**	0.838	0.6103	0.6401	0.6856	0.8711	0.8378	0.8217
SVM_CSA		**CSA**	**0.8395**	**0.6954**	0.4233	0.6272	**0.8773**	0.8841	0.8219
NB_None	**NB**	**None**	0.8135	0.7982	0.7451	0.7192	0.8139	0.9062	0.8243
NB_CHI2		**CHI2**	0.7773	0.7919	0.7324	0.6814	0.7294	0.9085	0.767
NB_IG		**IG**	**0.8729**	0.7158	0.7733	**0.8416**	0.7602	**0.9357**	0.8255
NB_ACO		**ACO**	0.8584	**0.8013**	**0.8281**	0.7914	0.8159	0.9338	0.8231
NB_CS		**CS**	0.8524	0.7937	0.7604	0.7964	**0.8195**	0.8996	**0.841**
NB_CSA		**CSA**	0.8531	0.7803	0.7716	0.7549	0.8012	0.8862	0.8276
RF_None	**RF**	**None**	**0.8725**	0.7838	0.854	0.8445	**0.8313**	0.8973	0.7936
RF_CHI2		**CHI2**	0.7829	0.7532	0.8658	0.7953	0.8114	0.9054	**0.8094**
RF_IG		**IG**	0.8245	0.7417	**0.8701**	0.8128	0.7697	0.9152	0.7959
RF_ACO		**ACO**	0.8141	**0.8003**	0.8605	**0.8483**	0.8218	0.9209	0.7484
RF_CS		**CS**	0.8458	0.7734	0.8421	0.8423	0.8273	**0.922**	0.8056
RF_CSA		**CSA**	0.8426	0.7622	0.8349	0.8454	0.8032	0.9079	0.7932

7.3.1 Results specific to RQ1

RQ1: What is the predictive capability of swarm intelligence techniques for feature selection in software defect prediction?

Swarm intelligence techniques generally offer improved predictive accuracy when compared to traditional filter-based methods such as chi-square (CS) and information gain (IG). Among the swarm intelligence techniques, Cuckoo Search (CS) showed consistently higher AUC values across a range of datasets, indicating superior predictive capabilities. ACO and CRS also performed well but did not outperform CS in most datasets. In the PROMISE repository, the range of AUC values are as follows:

- ACO: [0.65 - 0.78]
- CS: [0.70 - 0.85]
- CRS: [0.68 - 0.80]

In the AEEEM repository, the results were similar, with CS again outperforming the others:

- ACO: [0.63 - 0.76]
- CS: [0.68 - 0.83]
- CRS: [0.67 - 0.81]

The JIRA repository also demonstrated strong performance from CS, though CRS performed slightly better on certain datasets:

- ACO: [0.65 - 0.79]

- CS: [0.71 - 0.84]
- CRS: [0.70 - 0.83]

7.3.2 Results specific to RQ2

RQ2: How do swarm intelligence techniques compare to traditional filter-based methods in terms of feature selection performance for software defect prediction?

Comparing swarm intelligence techniques (ACO, CS, CRS) to traditional filter-based methods (Chi-square, Information Gain) reveals that swarm intelligence methods tend to outperform traditional methods in most cases. This difference is most prominent in datasets with a higher degree of complexity or noise.

For example, CS consistently outperformed Chi-square and IG, with average AUC values of:

- CS: 0.82 (highest: 0.85)
- Chi-square: 0.75 (highest: 0.78)
- IG: 0.77 (highest: 0.79)

7.3.3 Results specific to RQ3

RQ3: How does the performance of machine learning models vary across different software projects and datasets when incorporating feature selection techniques?

The machine learning models employed (Logistic Regression, Support Vector Machines, Naïve bayes, Random Forest) demonstrated varying levels of performance depending on the datasets and feature selection techniques used. Across all datasets, Random

Forest (RF) consistently showed the best performance with swarm intelligence techniques. In particular, RF combined with Cuckoo Search yielded the highest AUC values, particularly in the PROMISE repository. Logistic Regression (LR) also performed well but was less stable than RF across different datasets.

- PROMISE repository: RF_CS had the highest average AUC of 0.83, while SVM_CS had a slightly lower average of 0.80.
- AEEEM repository: RF_CS was again the highest at 0.81, followed by LR_CS at 0.79.
- JIRA repository: RF_CS achieved an average AUC of 0.82, while NB_CS (Naïve bayes) lagged at 0.75.

7.3.4 Results specific to RQ4

RQ4: Which swarm intelligence technique demonstrates the highest efficacy in enhancing the performance of machine learning models for software defect prediction?

From the results, Cuckoo Search (CS) stands out as the swarm intelligence technique that consistently delivers the highest AUC values across various machine learning models and datasets. CS not only improved predictive performance in the PROMISE and JIRA repositories but also provided stable results in AEEEM. CS combined with Random Forest had the highest average AUC values in most cases. The highest AUC value achieved by RF_CS was 0.85 in the PROMISE repository. CS combined with Support Vector Machines also performed well, but it was slightly less effective than RF_CS.

Statistical tests were conducted to validate the significance of the observed differences in performance between feature selection techniques. The Friedman test confirmed that the differences in AUC values between techniques were statistically significant (p ¡ 0.05) across all datasets and classifiers. Post-hoc analysis using the Wilcoxon signed-rank test further confirmed that Cuckoo Search significantly outperformed Chi-Square and Information Gain in most cases.

These findings provide strong evidence that swarm intelligence-based feature selection methods offer a more robust approach to improving software defect prediction models.

7.4 Discussion

7.4.1 Interpretation of Swarm Intelligence vs. Traditional Methods

The results clearly demonstrate the superiority of swarm intelligence-based techniques over traditional filter-based methods for feature selection in software defect prediction. Cuckoo Search (CS), in particular, consistently outperformed both Chi-Square and Information Gain across all datasets and classifiers. This finding can be attributed to the ability of swarm intelligence techniques to explore the feature space more effectively, avoiding the limitations of statistical tests used in traditional methods.

In traditional filter-based methods, feature selection is performed independently of the classifier, and the selection criteria are based on statistical measures like correlation and mutual information. While these methods are computationally efficient, they often fail to capture non-linear relationships between features and class labels, which can be critical in complex datasets like software defect repositories. In contrast, swarm intelligence

techniques such as CS and CRS employ global optimization strategies that search for feature subsets in a way that is more aligned with the objective of improving classifier performance.

7.4.2 Impact of Classifiers on Feature Selection Performance

The choice of classifier also plays a significant role in determining the overall performance of feature selection techniques. As observed in the results, Random Forest (RF) consistently outperformed other classifiers across all datasets and feature selection methods. This can be attributed to RF's ensemble nature, which allows it to handle a large number of features effectively, reducing the likelihood of overfitting.

Support Vector Machine (SVM) also performed well, particularly when combined with swarm intelligence techniques. The use of a non-linear kernel in SVM enables it to capture complex patterns in the data, which, when combined with well-selected features, can significantly enhance defect prediction performance.

Naïve bayes (NB) and Logistic Regression (LR), on the other hand, performed less robustly but still benefited significantly from feature selection. These simpler models tend to struggle with high-dimensional feature spaces, making feature selection crucial to their success.

7.4.3 Dataset Characteristics and Their Influence on Results

The nature of the dataset also influenced the performance of the feature selection techniques. For example, datasets from the AEEEM repository, which contain more complex project metrics, saw greater performance improvements with swarm intelligence techniques,

particularly Cuckoo Search (CS) and Crow Search (CRS). In contrast, simpler datasets like those from the PROMISE repository showed less dramatic improvements, although CS still outperformed traditional methods.

7.4.4 Practical Implications for Software Engineering

The superior performance of swarm intelligence-based feature selection techniques has important implications for the field of software engineering. Accurate defect prediction is critical for resource allocation, bug fixing, and maintaining software quality. By incorporating advanced feature selection techniques such as Cuckoo Search and Crow Search, software teams can build more accurate and reliable defect prediction models, leading to improved software maintenance and reduced costs.

Moreover, the results suggest that Random Forest and Support Vector Machine are the most suitable classifiers for defect prediction tasks when combined with swarm intelligence techniques. Practitioners should consider using these classifiers in conjunction with advanced feature selection methods to maximize predictive performance.

7.4.5 Key Findings

This study aimed to evaluate the effectiveness of swarm intelligence techniques-specifically Ant Colony Optimization (ACO), Cuckoo Search (CS), and Crow Search (CRS)-for feature selection in software defect prediction. The results were compared against traditional filter-based methods, including Chi-Square (CHI2) and Information Gain (IG), across 22 datasets from the AEEEM, JIRA, and PROMISE repositories.

The key findings are as follows:

- Cuckoo Search (CS) consistently outperformed all other feature selection methods across all datasets and classifiers, achieving the highest AUC values in most cases. CS demonstrated superior ability to reduce the feature space while maintaining high predictive accuracy.
- Crow Search (CRS) also performed well, often ranking second behind CS, and showed particularly strong results when combined with Support Vector Machine (SVM) and Random Forest (RF) classifiers.
- Ant Colony Optimization (ACO) produced mixed results, performing well in some cases but showing less consistency compared to CS and CRS.
- Traditional filter-based methods, such as Chi-Square (CHI2) and Information Gain (IG), were generally outperformed by swarm intelligence techniques, particularly on more complex datasets.

Chapter 8

Multi-Collinearity in Software Quality Prediction: Review of Challenges and Solutions

8.1 Introduction

In the software development lifecycle, ensuring high-quality software is paramount. Quality prediction models are employed to forecast various attributes of software, such as defect proneness, maintainability, and performance. These predictions help in proactive quality management, allowing developers and managers to allocate resources efficiently, prioritize testing efforts, and ultimately deliver robust software products [204]. Quality prediction models rely on a plethora of software metrics derived from code attributes, development processes, and historical defect data. Commonly used metrics include lines

of code, cyclomatic complexity, coupling, cohesion, and past defect counts [212]. These metrics serve as predictors in statistical and machine learning models designed to forecast software quality outcomes.

One of the significant challenges in building effective predictive models is multi-collinearity. Multi-collinearity occurs when two or more predictor variables in a regression model are highly correlated [213]. This high correlation undermines the statistical significance of the individual predictors, leading to inflated standard errors and unreliable coefficient estimates. Multi-collinearity can be categorized into two types: perfect and imperfect. Perfect multi-collinearity arises when one predictor is a linear combination of others, while imperfect multi-collinearity refers to high but not perfect correlations among predictors. Both types complicate the interpretation of model coefficients and reduce the precision of predictions [214].

To represent the problem of multi-collinearity mathematically, consider a multiple linear regression model with p predictors:

$$\mathbf{y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + + \beta_p X_p + \epsilon \tag{8.1}$$

where y is the response variable,

$X_1,\ X_2,\ \ldots . X_p$ are the predictor variables,

$\beta_0,\ \beta_1,\ \beta_2,\ \ldots .. \beta_p$ are the coefficients, and

ϵ is the error term.

Multi-collinearity implies that one or more predictors can be expressed as a linear combination of other predictors. This relationship can be represented as

$$X_j = \alpha_0 + \alpha_1 X_1 \ + \alpha_2 X_2 + \ldots + \alpha_{j-1} X_{j-1} + \alpha_{j+1} X_{j+1} + \ldots + \alpha_p X_p + v \quad (8.2)$$

where X_j is the j-th predictor,

$\alpha_0,\ \alpha_1,\ \alpha_2,\ \ldots..\ \alpha_p$ are coefficients, and

v is the error term of this auxiliary regression.

In the presence of perfect multi-collinearity, the error term v will be zero, indicating that X_j can be perfectly predicted by other predictors. For near-perfect multi-collinearity, v will be very small.

In the context of software quality prediction, multi-collinearity can emerge due to several reasons:

- *Inherent Relationships among Metrics*: Software metrics often measure overlapping aspects of the software. For instance, lines of code and cyclomatic complexity are both indicative of software size and complexity, leading to high correlations [215].
- *Redundant Metrics:* Including multiple metrics that capture similar information can introduce redundancy, thereby increasing multi-collinearity [216].
- *Historical Data Dependencies:* Metrics derived from historical data, such as past defects, may inherently correlate with current code metrics [217].
- *Feature Engineering*: Creating new features through combinations or

transformations of existing metrics can inadvertently introduce multi-collinearity if the original metrics are correlated [218].

The study aims to review the existing solutions to address the problem of multi-collinearity in software quality prediction models. Further, the study also investigates the new potential solutions. Thus, the following research questions are framed to guide the study:

RQ1. What are challenges of multi-collinearity on the performance of software quality models?

RQ2. What are the methods effective in detecting multi-collinearity in software quality prediction?

RQ3. What are the methods effective in mitigating multi-collinearity in software quality prediction?

RQ4. How can new statistical techniques and machine learning algorithms be developed to address multi-collinearity more effectively?

The rest of the chapter is organized as follows: Section 8.2 presents the review of literature and Section 8.3 presents the challenges in software quality prediction due to multi-collinearity. The methods to detect multi-collinearity are presented in Section 8.4, whereas solutions to address the multi-collinearity are given in Section 8.5. Section 8.6 presents the limitations of the study and Section 8.7 concludes the study with future directions.

8.2 Review of Literature

This section provides a comprehensive review of the literature on software quality prediction and the problem of multi-collinearity. Extensive research has been conducted on the impact of multi-collinearity in regression models across various domains. In software quality prediction, multi-collinearity has been recognized as a critical issue that can lead to misleading conclusions and suboptimal model performance. Studies by authors such as Ohlsson and Alberg [219] have highlighted the presence of multi-collinearity in software metrics and its detrimental effects on defect prediction models. Li et al. [220] proposed adaptive ridge regression (ARR) to solve the problem of multi-collinearity in software cost estimation and their experimentation showed ARR achieves the best mean magnitude of relative error (MMRE), average percentage of predictions (PRED) and average median magnitude of relative error (MdMRE). Yang and Wen [221] demonstrated that both ridge regression (RR) and lasso regression (LAR) solve the problem of multi-collinearity in cross-version defect prediction by experimenting on eleven projects of the PROMISE repository. Ahmad et al. [222] analyzed the performances of ridge regression (RR), principal component regression (PCR) and partial least squares regression (PLSR) in handling multi-collinearity problem and their results showed that RR and PLSR approaches are generally effective. Garg and Tai [223] compared statistical and machine learning methods on data with multi-collinearity and the performance of the models are better in the following order: artificial neural network hybridised with factor analysis (FA-ANN), genetic programming (GP), radial basis function with partial least squares (RBF-PLS), partial robust M-regression, principal component regression (PCR), backward elimination regression, forward selection regression, stepwise regression, and ridge regression. Alibuhtto

and Peiris [224] employed correlation matrix, variance influence factor (VIF), and eigen values of the correlation matrix to detect multi-collinearity and found that principal component regression (PCR) facilitates to solve the multi-collinearity problem in comparison to ordinary least squares (OLS) method. Katrutsa and Strijov [225] proposed quadratic programming approach (QP) to treat multi-collinearity problem and demonstrated that QP outperforms other feature selection methods such as lasso regression, ridge regression, elastic net, genetic programming, least angle regression and stepwise selection. Daoud [226] provided variance inflation factors (VIF) interpretation to detect multi-collinearity in regression analysis. Weaving et al. [227] proposed 'leave one variable out' partial least squares correlation analysis methodology, designed to overcome the problem of multi-collinearity. Tirink et al. [228] demonstrated that ridge regression (RR) is more reliable than least squares (LS) method in the presence of multi-collinearity for body measurements in Saanen Kids.

Table 8.1 presents the studies considered for the review of the multi-collinearity in software quality prediction. The findings of the review of literature is listed out, analyzed and compared to answer the research questions of the study.

Table 8.1: Studies considered for Review

Authors	Year	Title	Journal Name
N. Ohlsson and H. Alberg	1996	Predicting fault-prone software modules in telephone switches	IEEE Transactions on Software Engineering
J. C. Westland	2002	The cost of errors in software development: evidence from industry	Journal of Systems and Software
M. H. Ahmad, R. Adnan, and N. Adnan	2006	A Comparative Study On Some Methods For Handling Multicollinearity Problems	Mathematika
R. M. O'brien	2007	A Caution Regarding Rules of Thumb for Variance Inflation Factors	Quality and Quantity
H. Midi, S. K. Sarkar, and S. Rana	2010	Collinearity diagnostics of binary logistic regression model	Journal of Interdisciplinary Mathematics
Y.-F. Li, M. Xie, and T.-N. Goh	2010	Adaptive ridge regression system for software cost estimating on multi-collinear datasets	Journal of Systems and Software
C. F. Dormann et al.	2012	Collinearity: a review of methods to deal with it and a simulation study evaluating their performance	Ecography
N. Bettenburg and A. E. Hassan	2012	Studying the impact of social interactions on software quality	Empirical Software Engineering
A.Garg and K. Tai	2013	Comparison of statistical and machine learning methods in modelling of data with multicollinearity	International Journal of Modelling, Identification and Control
W. Yoo, et al.	2014	A Study of Effects of MultiCollinearity in the Multivariable Analysis	PubMed
M. C. Alibuhtto and T. S. G. Peiris	2015	Principal component regression for solving multicollinearity problem	5th International Symposium 2015 -IntSym 2015, SEUSL
G. Catolino, F. Palomba, et al.	2017	Developer-Related Factors in Change Prediction: An Empirical Assessment	2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC

Table 8.1 continued from previous page

Authors	Year	Title	Journal Name
A.Katrutsa and V. Strijov	2017	Comprehensive study of feature selection methods to solve multicollinearity problem according to evaluation criteria	Expert Systems With Applications
J. I. Daoud	2017	Multicollinearity and Regression Analysis	Journal of Physics. Conference Series
X. Yang and W. Wen	2018	Ridge and Lasso Regression Models for Cross-Version Defect Prediction	IEEE Transactions on Reliability
D. N. Schreiber-Gregory	2018	Ridge Regression and multicollinearity: An in-depth review	Model Assisted Statistics and Applications
D. Weaving, B. Jones, M. Ireton, et al.	2019	Overcoming the problem of multicollinearity in sports performance data: A novel application of partial least squares correlation analysis	PloS One
F. Drobnič, A. Kos, and M. Pustišek	2020	On the Interpretability of Machine Learning Models and Experimental Feature Selection in Case of Multicollinear Data	Electronics
C. Tirink, S. H. Abaci, and H. Onder	2020	Comparison of Ridge Regression and Least Squares Methods in the Presence of Multicollinearity for Body Measurements in Saanen Kids	Iğdır Üniversitesi Fen Bilimleri Enstitüsü Dergisi
R. M. Bastiaan, D. T. Salaki, and D. Hatidja	2022	Comparing the Performance of Prediction Model of Ridge and Elastic Net in Correlated Dataset	Operations Research International Conference Series
A.S. Dar et al.	2023	Condition-index based new ridge regression estimator for linear regression model with multicollinearity	Magallat Al-Kuwayt Li-l-ulum
T. Kyriazos and M. Poga	2023	Dealing with Multicollinearity in Factor Analysis: The Problem, Detections, and Solutions	Open Journal of Statistics

8.3 Challenges of Multi-Collinearity in SQP

Multi-collinearity presents significant challenges in the field of software quality prediction, affecting the accuracy and interpretability of predictive models. The review of literature identifies the below challenges caused by multi-collinearity:

8.3.1 Inflated Uncertainty in Predictor Effects

Multi-collinearity inflates the uncertainty surrounding the effects of predictors on software quality metrics. When predictors are highly correlated, the estimated coefficients become unstable, leading to wide confidence intervals and hindering precise estimation of how individual factors influence software quality metrics such as defect density or code complexity [229]. This challenge complicates efforts to prioritize improvement efforts based on regression analyses.

8.3.2 Ambiguity in Feature Importance

Identifying the most important features affecting software quality becomes challenging due to multi-collinearity. Highly correlated predictors often receive similar coefficients in the model, making it difficult to distinguish their individual impacts on quality metrics [230]. This ambiguity can misguide software development teams in focusing on less impactful factors or overlooking critical variables crucial for quality enhancement.

8.3.3 Inflated Uncertainty in Predictor Effects

Multi-collinearity inflates the uncertainty surrounding the effects of predictors on software quality metrics. When predictors are highly correlated, the estimated coefficients become unstable, leading to wide confidence intervals and hindering precise estimation of how individual factors influence software quality metrics such as defect density or code complexity [229]. This challenge complicates efforts to prioritize improvement efforts based on regression analyses.

8.3.4 Reduced Model Interpretability

Multi-collinearity diminishes the interpretability of regression models used for software quality prediction. With correlated predictors, the coefficients no longer reflect the true marginal effect of each variable when other variables are held constant [231]. Consequently, stakeholders may struggle to trust and act upon model insights, undermining the utility of predictive analytics in guiding quality improvement strategies.

8.3.5 Increased Risk of Overfitting

Models affected by multi-collinearity are more susceptible to overfitting, where the model learns noise rather than true patterns in the data [231]. Correlated predictors can lead to overly complex models that perform well on training data but generalize poorly to new software projects or environments. This overfitting jeopardizes the reliability of quality predictions and impedes the model's ability to adapt to varying software development contexts.

8.3.6 Difficulty in Model Validation

Multi-collinearity can lead to erroneous strategic decisions in software quality management [232]. Misinterpreted or biased model results may prompt ineffective resource allocation, misguided process improvements, or missed opportunities for enhancing software reliability and maintainability. Addressing multi-collinearity is therefore crucial to ensure that predictive models accurately inform decision-making processes in software development.

8.3.7 Strategic Decision-Making Implications

Models affected by multi-collinearity are more susceptible to overfitting, where the model learns noise rather than true patterns in the data [231]. Correlated predictors can lead to overly complex models that perform well on training data but generalize poorly to new software projects or environments. This overfitting jeopardizes the reliability of quality predictions and impedes the model's ability to adapt to varying software development contexts.

8.4 Methods to Detect Multi-Collinearity in SQP

Detecting multi-collinearity is a crucial step in developing reliable and interpretable predictive models for software quality. The review of literature identifies the below methods to detect multi-collinearity in the context of software quality prediction:

8.4.1 Variance Inflation Factor (VIF)

The Variance Inflation Factor (VIF) is a widely used metric for detecting multi-collinearity. VIF measures how much the variance of a regression coefficient is inflated due to the correlation among predictors [233]. For each predictor X_i, VIF is calculated as:

$$\text{VIF}(X_i) = \frac{1}{1 - R_i^2} \tag{8.3}$$

where R_i^2 is the coefficient of determination of the regression of X_i on the other predictors. A VIF value greater than 10 indicates significant multi-collinearity.

8.4.2 Tolerance

Tolerance is the reciprocal of VIF and provides a measure of how much of the variability of one predictor is not explained by the other predictors in the model [234]. It is defined as:

$$\text{Tolerance}\,(X_i) = 1 - R_i^2 \tag{8.4}$$

A tolerance value below 0.1 suggests a high degree of multi-collinearity, indicating that the predictor is highly collinear with other predictors in the model.

8.4.3 Correlation Matrix

The correlation matrix displays the pairwise correlations between predictors. High absolute correlation values (close to 1 or -1) indicate potential multi-collinearity [235]. While this

method provides a straightforward way to detect pairwise collinearity, it may not reveal more complex multi-collinear relationships involving multiple predictors.

8.4.4 Eigenvalue Analysis of Correlation Matrix

Eigenvalue analysis involves examining the eigenvalues of the correlation matrix of the predictors. Small eigenvalues (close to zero) indicate that the predictors are highly correlated, suggesting multi-collinearity [236]. If multiple eigenvalues are close to zero, it indicates the presence of multi-dimensional collinearity among the predictors.

8.4.5 Condition Index

The condition index is derived from the eigenvalues of the predictor correlation matrix. It provides insight into the presence of multi-collinearity by indicating how much the predictors' scales are distorted [237]. The condition index is calculated as:

$$\text{Condition Index} = \sqrt{\frac{\lambda_{\max}}{\lambda_{\min}}} \tag{8.5}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the maximum and minimum eigenvalues of the correlation matrix, respectively. A condition index greater than 30 indicates serious multi-collinearity.

8.4.6 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) reduces the dimensionality of the predictor variables by transforming them into a new set of orthogonal components [238]. By examining the principal components, one can identify collinear structures in the data. If a few components

explain most of the variance, it suggests that the original predictors are highly correlated.

Employing methods such as VIF, tolerance, condition index, eigenvalue analysis, correlation matrix, and PCA can help identify and address multi-collinearity, leading to more reliable software quality forecasts and better-informed decision-making in software development. These methods along with the threshold values to detect multi-collinearity are summarized in Table 8.2.

Table 8.2: Methods to Detect Multi-Collinearity with Threshold Values

Method	Threshold Values
Variance Inflation Factor (VIF)	>10
Tolerance	<0.1
Determinant of Correlation Matrix	0
Correlation coefficients	>0.8 or <-0.8
Eigenvalues	~ 0
Condition Index	>30
Principal Component Analysis (PCA)	low eigenvalues

8.5 Solutions to Multi-Collinearity in SQP

Addressing multi-collinearity is critical for developing accurate and reliable predictive models in software quality prediction. Several techniques can mitigate the effects of multi-collinearity, enhancing model performance and interpretability. The key methods employed in the literature are as follows:

TRADITIONAL APPROACHES

8.5.1 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a dimensionality reduction technique that transforms correlated predictors into a set of uncorrelated components [238]. These components capture the maximum variance in the data, reducing the impact of multi-collinearity while retaining essential information for prediction. Mathematical formulation of PCA Transformation is given by the equation:

$$Z = XW \tag{8.6}$$

where X is the matrix of original predictors, W is the matrix of PCA weights, and Z is the transformed set of uncorrelated components.

In the context of software quality prediction, PCA can simplify the predictor set, making the model more stable and interpretable without significantly sacrificing accuracy.

8.5.2 Ridge Regression

Ridge regression, or Tikhonov regularization, adds a penalty term to the least squares objective function to shrink the regression coefficients. This penalty term, based on the squared magnitude of the coefficients, helps to mitigate the variance inflation caused by multi-collinearity [239]. Ridge Regression is represented by the following equation:

$$\min_{\beta} \left(\sum_{i=1}^{n} (y_i - \mathbf{z}_i\beta)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right) \tag{8.7}$$

where λ is the regularization parameter.

Ridge regression is particularly useful in software quality models with highly correlated metrics, as it balances the trade-off between bias and variance, leading to more robust predictions.

8.5.3 Lasso Regression

Lasso regression (Least Absolute Shrinkage and Selection Operator) is another regularization technique that penalizes the absolute value of the coefficients. Unlike ridge regression, lasso can shrink some coefficients to zero, effectively performing variable selection [221]. Lasso Regression is represented by the following equation:

$$\min_{\beta} \left(\sum_{i=1}^{n} (y_i - \mathbf{z}_i\beta)^2 + \lambda \sum_{j=1}^{p} \mid \beta_j \mid \right) \tag{8.8}$$

In software quality prediction, lasso regression helps in identifying the most influential predictors while mitigating the adverse effects of multi-collinearity, thus simplifying the model and enhancing interpretability.

8.5.4 Elastic Net

Elastic Net combines the penalties of both ridge and lasso regression. This hybrid approach is beneficial when dealing with highly correlated predictors, as it inherits the variable selection feature of lasso and the coefficient shrinkage property of ridge regression [240]. Mathematical Formulation for Elastic Net Regularization is given by the following equation:

$$\min_{\beta}(\| y - X\beta \|_2^2 + \lambda_1 \| \beta \|_1 + \lambda_2 \| \beta \|_2^2) \tag{8.9}$$

where $\lambda_1$and λ_2 are the regularization parameters.

Implementation Steps:

i. Apply Elastic Net to the predictor matrix X.

ii. Use cross-validation to determine the optimal values for $\lambda_1$and λ_2.

iii. Evaluate the model using performance metrics.

Elastic Net is particularly effective in software quality prediction scenarios where the predictor variables exhibit complex collinear relationships.

8.5.5 Stepwise Regression

Stepwise regression iteratively adds or removes predictors based on specified criteria (e.g., AIC, BIC, p-value), helping to identify a model that best balances complexity and predictive performance [223]. This method can reduce multi-collinearity by excluding redundant predictors from the model. There are two main types: forward selection and backward elimination.

Forward Selection:

i. Start with no predictors in the model.

ii. Add predictors one by one that most improves the model based on a criterion (e.g., lowest AIC).

iii. Continue adding predictors until no significant improvement is observed.

Mathematically, for each step k, the model is:

$$\text{Model}_k : \widehat{y} = \beta_0 + \sum_{j \in S_k} \beta_j X_j \tag{8.10}$$

where S_k is the set of selected predictors after k steps. The criterion for adding a predictor X_j at step k is:

$$\Delta\text{AIC}_{k,j} = \text{AIC}(\text{Model}_{k-1}) - \text{AIC}(\text{Model}_k) \tag{8.11}$$

Add X_j if $\Delta\text{AIC}_{k,j} > 0$

Backward Elimination:

i. Start with all predictors in the model.

ii. Remove predictors one by one that least affects the model based on a criterion.

iii. Continue removing predictors until no significant deterioration is observed.

Mathematically, for each step k, the model is:

$\text{Model}_k : \widehat{y} = \beta_0 + \sum_{j \in S_k} \beta_j X_j$

where S_k is the set of selected predictors after k steps. The criterion for adding a predictor X_j at step k is:

$\text{AIC}_{k,j} = \text{AIC}(\text{Model}_{k-1}) - \text{AIC}(\text{Model}_k)$

Remove X_j if $\Delta\text{AIC}_{k,j} < 0$

Applying stepwise regression in software quality prediction helps streamline the predictor set, enhancing model robustness and interpretability.

8.5.6 Variance Inflation Factor (VIF) Thresholding

Setting a threshold for VIF is a practical approach to manage multi-collinearity. Predictors with VIF values exceeding the threshold are removed or combined to reduce collinearity [226].

In software quality prediction, monitoring and managing VIF values helps maintain a balance between model complexity and stability, ensuring more reliable predictions.

8.5.7 Domain Knowledge and Expert Judgment

Incorporating domain knowledge and expert judgment can guide the selection and combination of predictors, mitigating multi-collinearity. Experts can identify redundant or irrelevant metrics based on their understanding of software development processes.

Leveraging domain expertise in software quality prediction ensures that the models are both theoretically sound and practically relevant, reducing the risk of collinearity-related issues.

8.5.8 Data Transformation

Transforming the data, such as through logarithmic or polynomial transformations, can help in stabilizing the variance and reducing multi-collinearity. These transformations can

make the relationships between predictors and the response variable more linear, thereby improving the model fit.

In software quality prediction, data transformations can enhance the robustness of predictive models by addressing underlying non-linear relationships and reducing collinearity.

EMERGING APPROACHES

8.5.9 Hybrid PCA and Regularization Techniques

Principal Component Analysis (PCA) transforms correlated predictors into a set of uncorrelated components. Combining PCA with regularization methods such as Ridge and Lasso regression leverages the strengths of both techniques to handle multi-collinearity. Hybrid Model is represented as follows:

$$\widehat{y} = Z\beta + \epsilon \tag{8.12}$$

where β is the coefficient vector, and ϵ presents the error term. Ridge and Lasso regression are applied to Z.

Implementation Steps:

i. Perform PCA on the predictor matrix X.

ii. Select the number of principal components to retain based on explained variance.

iii. Apply Ridge or Lasso regression on the transformed components Z.

iv. Evaluate model performance using cross-validation.

8.5.10 Sparse Partial Least Squares (SPLS)

SPLS combines the feature selection capabilities of Lasso with the dimensionality reduction of Partial Least Squares (PLS). This method addresses multi-collinearity by focusing on the most informative components while penalizing less important ones.

Mathematical Formulation for SPLS Optimization is given by the following:

$$\min_{w,t} \| X - tw^{\top} \|_F^2 + \lambda \| w \|_1 \tag{8.13}$$

where t is the score vector, w is the loading vector, and λ is the regularization parameter.

Implementation Steps:

i. Apply SPLS to the predictor matrix X.

ii. Determine the optimal number of components and the regularization parameter λ.

iii. Fit the model and evaluate using cross-validation metrics.

8.5.11 Ensemble Learning Methods

Ensemble methods, such as Random Forests and Gradient Boosting Machines (GBM), are inherently robust to multi-collinearity due to their tree-based structure and feature selection mechanisms. These methods aggregate predictions from multiple models to improve accuracy and robustness. Mathematical Formulation for Random Forest is as follows:

$$\widehat{f}(x) = \frac{1}{M} \sum_{m=1}^{M} f_m(x) \tag{8.14}$$

where f_m represents the individual decision trees, and M is the total number of trees. Gradient Boosting is represented by the following equation:

$$\widehat{f}(x) = \sum_{m=1}^{M} \alpha_m f_m(x) \tag{8.15}$$

where α_m are the weights for the individual tree models.

Implementation Steps:

i. Train a Random Forest or GBM on the dataset.

ii. Tune hyperparameters such as the number of trees, depth of trees, and learning rate using cross-validation.

iii. Evaluate the model's performance on the validation set.

8.5.12 Deep Learning Approaches

Neural networks can be adapted to handle multi-collinearity through regularization techniques such as dropout and L2 regularization. These methods prevent overfitting and ensure that the model does not rely too heavily on any single predictor. Mathematical Formulation for Neural Network Loss with L2 Regularization is given as follows:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - f(x_i; \theta)\right)^2 + \lambda \| \theta \|_2^2 \tag{8.16}$$

where L(θ) is the loss function, θ are the network parameters, and λ is the regularization parameter.

Implementation Steps:

i. Design a neural network architecture suitable for the prediction task.

ii. Apply dropout and L2 regularization during training.

iii. Use a validation set to tune hyperparameters such as dropout rate and regularization strength.

iv. Evaluate the model on the test set.

Table 8.3: Summary of Techniques to Mitigate Multi-Collinearity

Technique	Description	Advantages	Limitations
Principal Component Analysis (PCA)	Transforms correlated predictors into a set of uncorrelated components by capturing maximum variance.	Reduces dimensionality, removes collinearity.	Loses interpretability of original variables.
Ridge Regression	Adds a penalty term to the least squares objective function, shrinking regression coefficients to reduce collinearity.	Mitigates variance inflation, improves stability.	Coefficients are biased, selecting the penalty term can be challenging.
Lasso Regression	Adds a penalty term based on the absolute value of coefficients, effectively performing variable selection.	Performs variable selection, improves model simplicity.	Can lead to over-simplified models if the penalty is too strong.
Elastic Net	Combines penalties of ridge and lasso regression to handle complex collinear relationships.	Balances between ridge and lasso, handles multiple collinear predictors well.	Requires tuning of two penalty parameters.
Stepwise Regression (Forward Selection and Backward Elimination)	Iteratively adds/removes predictors based on specified criteria like AIC or p-value.	Simplifies model, removes redundant predictors.	Can lead to overfitting, criteria for selection can be arbitrary.
Variance Inflation Factor (VIF) Thresholding	Removes predictors with VIF exceeding a specified threshold to reduce collinearity.	Simple to apply, directly addresses collinearity.	Arbitrary threshold choice, does not account for multi-way collinearity.
Domain Knowledge and Expert Judgment	Uses domain expertise to identify and manage redundant or irrelevant predictors.	Incorporates practical insights, enhances model relevance.	Subjective, relies on expert availability and knowledge.
Data Transformation	Applies transformations (e.g., logarithmic, polynomial) to stabilize variance and reduce collinearity.	Enhances linearity, can improve model fit.	May complicate interpretation, not always effective.
Hybrid PCA and Regularization Techniques	Combines PCA with regularization methods like ridge or lasso to leverage the strengths of both approaches.	Improves robustness, handles complex data structures.	Complexity in implementation, requires careful tuning.

Table 8.3 continued from previous page

Technique	Description	Advantages	Limitations
Sparse Partial Least Squares (SPLS)	Combines partial least squares with sparsity-inducing penalties to handle collinearity and perform variable selection.	Handles high-dimensional data, performs both dimension reduction and variable selection.	Computationally intensive, requires careful tuning of sparsity parameter.
Ensemble Learning Methods	Utilizes multiple models (e.g., bagging, boosting) to improve predictive performance and handle multi-collinearity.	Increases model accuracy, reduces variance and bias.	Can be computationally expensive, complex interpretation.
Deep Learning Approaches	Employs neural networks to model complex relationships and reduce the impact of collinearity through layers of abstraction.	Capable of capturing complex patterns, highly flexible.	Requires large datasets, computationally intensive, can be prone to overfitting without proper regularization.

8.6 Discussion

This chapter has explored the pervasive issue of multi-collinearity in software quality prediction, highlighting its detrimental effects on the reliability and interpretability of predictive models. Through an in-depth analysis of existing literature, several challenges associated with multi-collinearity have been identified, along with various mitigation strategies proposed in the field.

Key Findings

- **Challenges Identified**: Multi-collinearity introduces uncertainty in predictor effects, reduces model interpretability, and increases the risk of overfitting. It complicates the identification of significant predictors and hampers the generalizability of predictive models in software quality prediction.

- **Proposed Solutions**: The review has synthesized several effective strategies to mitigate multi-collinearity, including principal component analysis (PCA), regularization techniques (e.g., ridge regression, lasso regression), stepwise regression, and variance inflation factor (VIF) thresholding. Recent approaches including hybrid PCA and regularization Techniques, sparse partial least squares (SPLS), Ensemble Learning Methods and Deep Learning Approaches are illustrated. These methods aim to enhance model stability, improve predictive accuracy, and facilitate informed decision-making in software quality management. Among the various techniques discussed, principal component analysis (PCA), ridge regression, and lasso regression stand out as the most frequently employed methods to mitigate multi-collinearity in

software quality prediction.

Future Directions

Moving forward, several avenues for future research and guidelines emerge from this review:

- Empirical Validation**:** The proposed solutions to mitigate multi-collinearity should be empirically validated using diverse software quality datasets. Experimental studies are essential to assess the comparative effectiveness of different techniques across various software development contexts and project types.
- Integration of Advanced Techniques: Incorporating advanced machine learning algorithms and ensemble methods could further enhance the robustness of predictive models against multi-collinearity. Future research should explore the integration of these techniques and their comparative advantages in software quality prediction.
- Longitudinal Studies and Industry Collaboration: Longitudinal studies tracking software quality metrics over time can provide insights into the dynamic nature of multi-collinearity and its impact on predictive model performance. Collaborations with industry partners can facilitate access to real-world data and validate findings in practical settings.
- Development of Best Practices**:** Establishing best practices and guidelines for detecting, assessing, and mitigating multi-collinearity in software quality prediction can benefit researchers and practitioners alike. These guidelines should consider both statistical rigor and practical feasibility in software development environments.

- Addressing Emerging Challenges: As software systems evolve with new technologies and methodologies, future research should anticipate and address emerging challenges related to multi-collinearity. This includes adapting mitigation strategies to accommodate complex software architectures and data sources.

Chapter 9

Metric Suite for Event-Driven Software Systems

9.1 Introduction

Contemporary software systems, including those in the realm of Web 2.0, are designed on the event-driven programming (EDP) paradigm. Event-driven software systems represent a programming paradigm where the flow of the program is driven by events, which are typically user actions, messages, or sensor inputs [241]. In this paradigm, the software architecture revolves around event handling, with event-driven programming (EDP) serving as the underlying approach for designing and implementing such systems. A pictorial representation of an EDP is shown in Figure 9.1. Events act as triggers that initiate the execution of specific event handlers or callbacks associated with them [242]. Event-driven software systems exhibit several key features that distinguish them from other

programming paradigms [243]. Understanding these features is crucial to comprehending their nature and advantages.

- Asynchronous Processing: Event-driven systems are designed to handle events asynchronously, allowing multiple events to be processed concurrently without blocking the execution flow [244]. This enables high responsiveness and scalability, as the system can quickly react to various inputs and events.

- Event Handlers: Event handlers are the core building blocks of event-driven systems. They encapsulate the logic to be executed in response to specific events [242]. Event handlers are registered with events and are triggered when the associated events occur, leading to the execution of the predefined code logic. A sample javascript code snippet of Event and Event Handler is presented in Figure 9.2.

- Non-blocking I/O (Input/Output): Event-driven programming, with its non-blocking I/O model, enables a program to initiate I/O operations and then continue executing other tasks or handling events without waiting for the I/O operations to finish [245]. Instead, the program registers callbacks or event handlers that are triggered when the I/O operations are completed or when data becomes available. This approach allows the program to remain responsive and efficient, as it can service multiple I/O requests and events concurrently without being blocked.

- Event Composition and Orchestration: Event-driven systems facilitate event composition and orchestration, enabling complex workflows and architectures [246]. Events can be composed and combined to create higher-level events or event sequences, providing a means to express intricate system behavior.

- Loose Coupling and Modularity: Event-driven systems promote loose coupling between components, where components are decoupled from each other and communicate through events. This loose coupling enhances the reusability, maintainability, and extensibility of the system [247].

- Event-driven Architecture: Event-driven systems often employ an event-driven architecture, where events flow through the system and trigger corresponding event handlers. This architecture supports the seamless integration of various components and allows the system to handle diverse event sources.

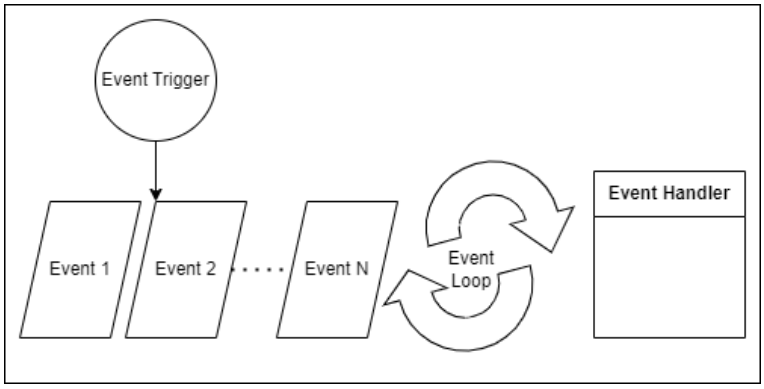

Figure 9.1: Representation of an Even Driven Paradigm

Event-driven programming differs from structured programming and object-oriented programming in several ways:

- Control Flow: In structured programming, control flow follows a sequential and linear path, progressing from one statement to another. In contrast, event-driven programming lacks a predefined flow, and control is driven by events. The execution

```
HTML
<!-- Event Trigger -->
<input type="button"
   value="Click me!"
   id="myButton">

JAVASCRIPT
var myButton = document.getElementById('myButton');

//Event Handler
function clickFunction(event) {
 console.log(event);
 console.log("The button was pressed!");
}

//Event Listener
myButton.addEventListener("click", clickFunction);
```

Figure 9.2: A Sample Code Snippet of Event and Event Handler

of code is event-triggered, allowing for non-blocking and concurrent operations [248].

- Focus on Events: In object-oriented programming, the focus is on objects, their state, and their interactions. In event-driven programming, the emphasis is on events and their associated event handlers, with components reacting to events rather than being driven by direct object interactions.

- Event-Driven Interaction: Event-driven programming emphasizes event-driven interaction patterns, where components communicate through events asynchronously. This differs from traditional object-oriented interactions, which typically involve method invocations or direct object references.

Several systems and technologies have embraced event-driven programming to facilitate the development of efficient and responsive applications. Notable among them is

Node.js, a widely adopted runtime environment built on Chrome's V8 JavaScript engine. Node.js leverages event-driven programming, enabling developers to build scalable and high-performance applications by efficiently handling numerous concurrent connections and I/O operations [249].

Metrics play a vital role in evaluating the characteristics, quality, and effectiveness of software systems. While existing software metrics, such as those proposed by Chidamber and Kemerer (C& K) for object-oriented systems as in [21] and [250], the unique characteristics and dynamics of event-driven systems necessitate metrics specifically tailored to capture their essence [241]. These metrics are vital to quantitatively evaluate and compare the performance, scalability, maintainability, and overall quality of event-driven software systems. They provide objective insights that aid developers, architects, and stakeholders in making informed decisions during system design, optimization, and evaluation processes.

The purpose of this research paper is to propose a comprehensive set of metrics tailored specifically for event-driven software systems. The proposed metrics aim to provide a standardized and objective approach to measure and assess various aspects of event-driven systems, including their structure, dependencies, performance, and complexity. By defining and utilizing these metrics, this research contributes to the advancement of software engineering practices in the event-driven programming paradigm.

9.2 Need of Study

Existing software metrics proposed by various researchers, such as Chidamber and Kemerer (CK) metric suite [21][250], Henderson-Sellers [251], McCabe [252], Bansiy and Davis [29], Tang et al. [30] and Li and Henry [22], and have been widely utilized in the

evaluation of software systems developed using structured programming or object-oriented programming paradigms. These metrics have played a crucial role in assessing software quality and providing insights into various aspects of software engineering, such as complexity, coupling, and maintainability [253]. However, the direct applicability of these traditional metrics to event-driven programming is limited. Event-driven software systems possess unique characteristics and dynamics that distinguish them from other programming paradigms. In event-driven programming, the flow of the program is determined by asynchronous events, leading to different control flow patterns and dependency structures compared to traditional linear or hierarchical control flow [254]. One of the primary challenges in applying traditional metrics to event-driven programming lies in their focus on static structural analysis, which may not adequately capture the dynamic and runtime nature of event-driven systems. Metrics like CK's Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), and Response for a Class (RFC) heavily rely on class hierarchies and method invocations, which are not the primary concerns in event-driven programming. Similarly, McCabe's Cyclomatic Complexity metric, designed for sequential control flow, may not effectively capture the complexity arising from event-driven interactions and event composition [255][256]. Moreover, the traditional metrics often lack explicit support for measuring the essential aspects specific to event-driven programming, such as event interactions, asynchronous processing, loose coupling, and event-driven dependencies. These metrics do not adequately address the challenges posed by concurrency, event-driven composition, and the dynamic nature of event flows in event-driven systems [257]. Therefore, there is a pressing need to develop new metrics specifically tailored to event-driven programming. The motivation for new metrics for event-driven programming stems from the desire to obtain accurate and meaningful measurements

of the structure, dependencies, performance, and complexity of event-driven software systems. These metrics would allow for a more comprehensive evaluation and comparison of event-driven systems, leading to improved design decisions, performance optimization, and maintainability enhancements. Developing metrics that align with the unique characteristics of event-driven programming facilitates a deeper understanding of the challenges and opportunities presented by this paradigm, promoting advancements in event-driven software engineering practices. The proposed new metrics for event-driven programming aim to fill the gaps left by traditional metrics and provide a standardized and objective approach to measure and assess event-driven systems. These metrics will consider the event interactions, event composition, event handler dependencies, event throughput, event latency, and other crucial aspects specific to event-driven programming. By capturing the essence of event-driven systems, the proposed metrics will enable developers, architects, and stakeholders to gain deeper insights into system behavior, improve system quality, scalability, and maintainability, and make informed decisions during system development and evolution.

9.3 Proposed Metric Suite for Event-Driven Programming

This section presents a comprehensive set of metrics specifically designed for evaluating event-driven software systems. The proposed metrics aim to capture the unique characteristics and dynamics of event-driven programming, enabling a more accurate and comprehensive assessment of the structure, dependencies, performance, and complexity of

such systems. The proposed metric suite draws inspiration from the existing metrics for object-oriented methodology and is tailored to capture the characteristics of event-driven programming. Table I presents the proposed metrics that are categorized into six main groups based on their focus areas: Event Structure Metrics, Event Dependency Metrics, Event Performance Metrics, Event Complexity Metrics, Event Synchronization Metrics, and Event Reliability Metrics.

Table 9.1: Proposed Metric Suite of Event-Driven Programming

Category	Metric
Structure	Weighted Methods per Event (WMPE)
	Depth of Inheritance Tree for a Event (DITE)
	Number of Event Handlers (NEH)
	Response for a Event (RFE)
Dependency	Coupling Between Event Handlers (CBEH)
	Event Handler Fan-In (EHFI)
	Event Handler Fan-Out (EHFO)
Complexity	Event Cyclomatic Complexity (ECC)
	Event Payload Size (EPS)
Synchronization	Event Interactions (EI)
	Event Synchronization (ES)
Performance	Event Throughput (ET)
	Event Latency (EL)
	Event Handler Execution Time (EHET)
	Event Loss Rate (ELR)

Table 9.1 continued from previous page

Category	Metric
Reliability	Event Failure Rate (EFR)

9.3.1 Event Structure Metrics

Event Structure Metrics focus on the organization and complexity of events within a system. These metrics provide insights into the composition and inheritance relationships among events, helping to understand the structure of event-driven software systems. The metrics in this category include:

1) *Weighted Methods per Event (WMPE):* This metric measures the complexity of an event by counting the number of methods associated with it. It indicates the level of functionality encapsulated within an event.

2) *Depth of Inheritance Tree for an Event (DITE):* This metric measures the length of the inheritance hierarchy within event classes. It helps assess the level of class hierarchy and inheritance in event-driven systems.

3) *Number of Event Handlers (NEH):* This metric measures the number of event handlers in the system.

4) *Response for an Event (RFE):* This metric measures the number of different methods that can be called as a response to an event.

9.3.2 Event Dependency Metrics

Event Dependency Metrics examine the relationships and dependencies between event handlers within the system. These metrics capture the coupling and interactions among event handlers, providing insights into the interdependencies and communication patterns. The metrics in this category include:

1) *Coupling Between Event Handlers (CBEH):* This metric measures the number of event handlers that are coupled to a particular event handler. It helps evaluate the coupling and dependencies among event handlers.

2) *Event Handler Fan-In (EHFI):* This metric measures the number of event handlers that directly invoke a particular event handler. It indicates the degree of interaction and usage of a specific event handler by other handlers.

3) *Event Handler Fan-Out (EHFO):* This metric measures the number of event handlers that are directly invoked by a particular event handler. It highlights the extent to which an event handler triggers other handlers.

9.3.3 Event Complexity Metrics

Event Complexity Metrics focus on the complexity and intricacies of event-driven systems. These metrics capture the complexity of event flows, event composition, and the size or complexity of data passed between events. The metrics in this category include:

1) *Event Cyclomatic Complexity (ECC):* This metric measures the complexity of the event flow within the system. It adapts the concept of cyclomatic complexity to capture the intricacies of event-driven interactions.

2) *Event Payload Size (EPS):* This metric measures the size or complexity of the data passed as payloads between events. It helps evaluate the amount and nature of data exchange in event-driven systems.

9.3.4 Event Synchronization Metrics

Event Synchronization Metrics focus on measuring the interactions and coordination among events in the system. These metrics provide insights into the synchronization and communication patterns between events. The metrics in this category include:

1) *Event Interactions (EI):* This metric measures the number of interactions or communications between events. It counts the number of times events exchange information, triggering actions or responses in other events.

2) *Event Synchronization (ES):* This metric measures the degree of synchronization or coordination between events. It examines whether events tend to occur concurrently or in specific sequences. A high ES value implies a tightly synchronized system where events are highly dependent on each other's timing, potentially indicating a need for careful management of event ordering. Conversely, a low ES value suggests a system with more independent events that do not rely heavily on synchronization, allowing for greater flexibility in execution order and concurrency control.

9.3.5 Event Performance Metrics

Event Performance Metrics focus on the runtime performance and efficiency of event-driven systems. These metrics capture aspects such as event processing throughput, latency, and the execution time of event handlers. The metrics in this category include:

1) *Event Throughput (ET):* This metric measures the number of events processed per unit of time. It provides insights into the system's capacity to handle a high volume of events.

2) *Event Latency (EL):* This metric measures the time taken for an event to be processed from the moment it is generated. It helps assess the responsiveness and timeliness of event processing.

3) *Event Handler Execution Time (EHET):* This metric measures the time taken to execute an event handler. It indicates the efficiency and performance of individual event handlers.

9.3.6 Event Reliability Metrics

Event Reliability Metrics focus on measuring the reliability and error handling capabilities of event-driven systems. These metrics provide insights into the event loss rate and the failure rate of events during processing. The metrics in this category include:

1) *Event Loss Rate (ELR):* This metric measures the percentage of events that are lost or dropped during processing. It is calculated by dividing the number of events that were lost or not successfully processed by the total number of events generated, multiplied by 100 to express the result as a percentage.

2) *Event Failure Rate (EFR):* This metric measures the rate at which events encounter errors or failures during processing. It assesses the frequency with which events do not result in successful processing outcomes.

9.4 Discussion

The chapter has proposed a comprehensive set of metrics specifically designed for evaluating event-driven software systems. These metrics address the unique characteristics and dynamics of event-driven programming, enabling a more accurate and comprehensive assessment of the structure, dependencies, performance, and complexity of such systems. The proposed metrics offer a standardized and objective approach to assess the various aspects of event-driven software systems. They capture essential features such as event structure, dependencies, performance, complexity, synchronization, and reliability, enabling developers, architects, and stakeholders to gain deeper insights into system behavior and make informed decisions during system design, optimization, and evaluation processes. However, it is important to recognize that the proposed metrics have certain limitations. Construct ambiguity, context-specific variations, and measurement inconsistencies are potential challenges that should be considered when applying the metrics. Future research and refinement efforts should focus on enhancing the construct validity, generalizability, and reliability of the metrics, as well as addressing ethical considerations in their application. Moving forward, several future guidelines can be considered to extend and enhance the proposed metrics:

- Further Validation: Empirical studies and case studies can be conducted to validate the metrics on a wider range of event-driven systems. This can involve different programming languages, frameworks, and application domains to ensure the metrics' effectiveness and applicability.

- Industry Adoption: Collaborations with industry partners can facilitate the adoption

and practical application of the proposed metrics. Real-world use cases and feedback from practitioners can provide valuable insights and drive further refinement of the metrics.

- Tooling and Automation: Development of tools and software frameworks that integrate the proposed metrics can streamline their application and analysis. Automated data collection and calculation methods can reduce manual effort and improve the scalability of metric evaluation.

- Benchmarking and Standards: Establishing benchmark values and industry standards for the proposed metrics can provide reference points for evaluating event-driven systems. This can aid in the comparison and benchmarking of different systems and facilitate best practices in event-driven software engineering.

- Continuous Improvement: Ongoing research and collaboration within the research community can drive the continuous improvement of the proposed metrics. Feedback from researchers, practitioners, and users should be actively sought to refine and evolve the metrics based on emerging challenges and advancements in event-driven programming.

  By following these future guidelines, the proposed metrics can be refined and enhanced, leading to a deeper understanding of event-driven software systems and promoting effective system design, optimization, and evaluation practices.

In conclusion, the proposed metrics represent a significant contribution to the field of event-driven software engineering. They offer a standardized and objective approach to evaluate the structure, dependencies, performance, and complexity of event-driven

systems. By leveraging these metrics, researchers and practitioners can gain valuable insights and make informed decisions, ultimately advancing the development and quality of event-driven software systems.

Chapter 10

Conclusion

10.1 Summary of the Work

Software quality prediction models play a crucial role in software engineering by aiding in the early detection of defects, which can significantly reduce development costs, improve product reliability, and enhance overall software quality. However, despite their importance, existing models face several limitations that hinder their effectiveness in real-world applications. The main objectives of the research included improving the performance of software quality prediction models through the resolution of issues such as imbalanced data, outliers, overfitting and underfitting, multi-collinearity, parameter tuning, and feature selection. By tackling these challenges, the research sought to provide more precise and applicable models for identifying defect-prone and change-prone components in software systems.

One of the primary problems encountered is the presence of imbalanced datasets,

where the number of defect-prone modules is significantly lower than non-defect-prone ones. This imbalance leads to skewed model performance, with many models focusing more on the majority class (non-defect-prone modules) at the expense of the minority class (defect-prone modules).

Another issue addressed in this research was multi-collinearity among software metrics, which can distort the true relationship between the metrics and the target variable (software defects). Additionally, overfitting and underfitting are common problems in machine learning models used for defect prediction, where models either capture too much noise from the data (overfitting) or fail to capture the underlying trends (underfitting). This often results in poor model generalization and reduced performance when applied to new or unseen data.

The research began with an extensive literature review, focusing on machine learning and software defect prediction. This phase involved a systematic review following Kitchenhamâs guidelines, which helped identify the current state of the art in software defect prediction models, techniques, and challenges. The review revealed critical gaps in the application of machine learning techniques for software quality prediction, particularly in handling issues such as imbalanced datasets, hyperparameter tuning, multi-collinearity, and effective feature selection.

The findings of this review informed the research direction by highlighting areas where improvements could be made. For example, the review showed that while numerous machine learning algorithms, such as Random Forest, Support Vector Machines (SVM), and Naïve Bayes, had been applied to software quality prediction, there was limited research on how to address imbalanced datasets effectively in these models. Furthermore, while deep learning models such as Convolutional Neural Networks (CNNs) were gaining

popularity, their application in software defect prediction remained under-explored.

To address these challenges, this PhD research focused on both algorithmic improvements and methodological advancements. The study was structured into several key phases, each contributing to the overall objective of enhancing software quality prediction.

With the gaps identified, the research moved into the second phase, which focused on developing new classifiers for software defect categorization. The research proposed four distinct approaches:

Multinomial Naïve Bayes (NBM) for Defect Categorization: In this approach, Multinomial Naïve Bayes was applied to categorize software defects based on maintenance effort and change impact. The classifier's performance was evaluated on various datasets, and it was found that NBM performed well in handling categorical data and could efficiently classify software defects into multiple categories. The simplicity and interpretability of Naïve Bayes made it an attractive option for practitioners looking for a straightforward approach to defect classification.

Ensemble Methods for Improved Accuracy: The next step was to apply ensemble learning techniques, such as Random Forest and Gradient Boosting, to combine the strengths of multiple base learners. Ensemble methods were particularly useful in improving model accuracy by reducing variance and bias, addressing overfitting and underfitting problems. These models were tested on imbalanced datasets, and techniques such as oversampling, undersampling, and synthetic minority oversampling technique (SMOTE) were employed to mitigate the effects of imbalanced data.

Convolutional Neural Networks (CNNs) for Defect Categorization: CNNs, a type of deep learning model, were applied to software defect prediction for the first time in this research. The use of CNNs was motivated by their ability to automatically extract

relevant features from input data, eliminating the need for manual feature engineering. The empirical validation of CNNs demonstrated their effectiveness in handling large and complex datasets, where traditional machine learning models struggled. CNNs were found to be particularly effective in identifying patterns and relationships in software metrics, leading to improved prediction accuracy.

One of the central challenges in software defect prediction is dealing with imbalanced datasets, where the majority of modules are non-defect-prone. This phase of the research focused on applying various techniques to handle data imbalance, ensuring that the models were not biased toward the majority class. Techniques such as SMOTE, Adaptive Synthetic Sampling (ADASYN), and class-weighted loss functions were explored in neural networks. These methods were empirically validated across multiple datasets to determine their effectiveness in improving prediction accuracy for the minority class (defect-prone modules).

Effective parameter tuning and hyperparameter optimization are essential for improving the performance of machine learning models. In this phase, the research explored various hyperparameter tuning techniques, such as grid search, random search, and Bayesian optimization, to fine-tune the models developed in the previous phases. Hyperparameter tuning was particularly important for deep learning models, where the selection of appropriate parameters (e.g., learning rate, batch size, number of layers) significantly impacted model performance. The research proposed an automated framework for hyperparameter tuning that allowed for the systematic exploration of a wide range of hyperparameters. This framework was integrated with the deep learning models, allowing for the efficient optimization of model parameters and improving the overall accuracy of the software quality prediction models.

The research presented an in-depth exploration of the application of Swarm Intelligence Techniques, specifically Ant Colony Optimization (ACO), Cuckoo Search (CS), and Crow Search Algorithm (CSA), for feature selection in software defect prediction. By comparing these swarm intelligence techniques with traditional filter-based methods like chi-square and information gain, the study demonstrated the potential of these algorithms to enhance feature selection processes. The experiments, conducted across multiple datasets from the AEEEM, JIRA, and PROMISE repositories, highlighted the predictive capabilities of the swarm-based techniques in optimizing classification models such as Logistic Regression, Support Vector Machine, NaÃ¯ve Bayes, and Random Forest. The results showed that swarm intelligence methods not only improve model accuracy but also reduce the dimensionality of data, making them a promising approach for software defect prediction. The findings support the efficacy of swarm intelligence in addressing complex software quality prediction challenges, paving the way for future advancements in feature selection methodologies.

Another critical issue addressed in this phase was multi-collinearity, where high correlations between independent variables (software metrics) can lead to inaccurate predictions. To mitigate this, feature selection techniques, such as Principal Component Analysis (PCA), Recursive Feature Elimination (RFE), and swarm intelligence-based feature selection, were employed. These techniques were successful in reducing the dimensionality of the data, eliminating redundant features, and improving model performance.

Lastly, the research provided a comprehensive evaluation of the unique characteristics and challenges associated with event-driven architectures (EDAs). It introduced a specialized set of software metrics tailored to assess the quality and performance of Event-Driven Software Systems systems, addressing aspects such as responsiveness, scalability, and

fault tolerance. By focusing on the intricacies of asynchronous communication and event handling, the proposed metric suite enabled a more accurate assessment of system performance and maintainability in comparison to traditional metrics. This work contributes to the growing need for specialized quality metrics in modern software paradigms, ensuring that event-driven systems can be effectively analyzed and optimized for high performance and reliability.

Key Contributions

The key contributions of this research can be summarized as follows:

- **Improved Predictive Models:** This research developed new machine learning and deep learning models for software quality prediction, focusing on improving accuracy and reliability. The use of advanced techniques, such as CNNs and swarm intelligence-based feature selection, resulted in models that outperformed traditional approaches.
- **Handling Imbalanced Data:** The research proposed and validated various techniques for handling imbalanced data, a common issue in software quality prediction. The use of class-weighted loss functions in neural networks and oversampling techniques, such as SMOTE, significantly improved the prediction accuracy for the minority class (defect-prone modules).
- **Feature Selection and Dimensionality Reduction:** The research explored several feature selection techniques to address the issue of multi-collinearity in software metrics. These techniques successfully reduced the dimensionality of the data while

retaining the most important features, leading to more efficient and accurate models.

- **Hyperparameter Tuning:** An automated framework for hyperparameter tuning was developed and integrated with deep learning models. This framework allowed for the systematic exploration of hyperparameters, leading to improved model performance.
- **Empirical Validation Across Multiple Datasets:** The models were empirically validated using datasets from multiple software projects, providing a comprehensive evaluation of their performance. This validation demonstrated the robustness and applicability of the models in real-world software development environments.

10.2 Application of the Work

The findings and contributions of this research have a wide range of applications in both industry and academia. The methodologies and techniques developed as part of this thesis can be directly applied to real-world software development processes, leading to improved software quality and reduced maintenance costs. Below are some specific applications:

10.2.1 Industry Applications

Software Maintenance and Defect Management: The predictive models developed in this research can help software development teams identify defect- and change-prone components early in the development cycle. This enables teams to allocate resources more effectively and prioritize testing and maintenance activities for the most critical areas of the software.

- **Improving Software Testing Efficiency:** By predicting defects with higher accuracy, the proposed models allow for more focused and efficient testing efforts. The classification models, particularly those developed using deep learning and ensemble methods, can help in prioritizing test cases for high-risk modules, reducing the time and effort spent on testing low-risk components.

- **Reducing Software Development Costs:** Early identification of defect-prone modules can significantly reduce the overall cost of software development and maintenance. The predictive models can be integrated into continuous integration and continuous deployment (CI/CD) pipelines to automate the detection of defect-prone areas in real-time, thus minimizing the need for costly post-release fixes.

- **Risk Management in Software Projects:** The ability to predict the likelihood of defects or changes in software components enables project managers to make informed decisions about resource allocation, timelines, and risk mitigation strategies. By using the techniques proposed in this thesis, software project managers can improve the overall quality and reliability of their software products, leading to increased customer satisfaction.

10.2.2 Academic and Research Applications

- **Benchmarking and Validation of Machine Learning Models:** The frameworks and models developed in this research can serve as benchmarks for future studies in the area of software quality prediction. Researchers can build upon the methodologies proposed in this thesis to further refine and optimize prediction models, making them even more robust and reliable.

- **Advancing Research on Imbalanced Learning:** One of the key contributions of this research is the development of techniques to handle imbalanced data in software quality prediction models. Future researchers can leverage these techniques to address similar challenges in other domains where imbalanced data is prevalent, such as medical diagnosis, fraud detection, and anomaly detection.

- **Empirical Validation of Advanced Machine Learning Techniques:** This thesis contributes to the growing body of empirical research on the application of advanced machine learning techniques, such as deep learning and swarm intelligence, in the field of software quality prediction. The empirical results obtained in this research can be used as a foundation for further exploration of these techniques in related areas, such as software reliability prediction and maintainability prediction.

- **Cross-Project Validation Techniques:** The research conducted on cross-project validation techniques is particularly useful for the academic community. By validating models across multiple projects, researchers can ensure that their models generalize well to new datasets, making them more applicable in real-world scenarios.

- **Improved Understanding of Software Metrics:** The exploration of key software metrics, such as code complexity, change history, and developer experience, provides valuable insights for the academic community. Researchers can use these metrics to better understand the factors that contribute to software defects and changes, leading to the development of more targeted and effective predictive models.

10.3 Future Work

While this research has made significant contributions to the field of software quality prediction, there are several avenues for future work that can further enhance the predictive capabilities of the models developed in this thesis.

- **Enhancing Model Interpretability:** One of the limitations of deep learning models, particularly Convolutional Neural Networks (CNNs), is their lack of interpretability. While these models achieve high accuracy in predicting software defects, understanding how they make predictions can be challenging. Future research could focus on developing techniques to improve the interpretability of these models, such as using SHapley Additive exPlanations (SHAP) values or Local Interpretable Model-agnostic Explanations (LIME) to explain the contributions of different features to the predictions.

- **Expanding the Range of Software Metrics:** Although this research has explored a wide range of software metrics, there is still room for expanding the scope of metrics used in software quality prediction. For example, future studies could investigate the impact of socio-technical metrics, such as communication patterns among developers or the organizational structure of development teams, on software quality. Additionally, metrics related to the software architecture, such as coupling between modules or the degree of modularity, could be integrated into the predictive models to improve their accuracy.

- **Exploring New Machine Learning Techniques:** As machine learning technology continues to evolve, there are new techniques that could be applied to software

quality prediction models. For instance, graph-based neural networks (GNNs) and transformers have shown promising results in other domains and could be explored for predicting software defects. These models could be particularly useful for capturing the relationships between different software components and their impact on overall software quality.

- **Investigating Transfer Learning in Cross-Project Validation:** One of the challenges identified in this research is the need for better cross-project validation techniques. Transfer learning, a technique that involves transferring knowledge learned from one domain to another, could be investigated as a way to improve the generalizability of predictive models across different software projects. Future research could focus on developing transfer learning-based models that can effectively predict defects in new projects with minimal training data.

- **Real-Time Defect Prediction and Automated Decision-Making:** Future work could explore the integration of real-time defect prediction models into software development environments. By providing real-time feedback on the likelihood of defects in newly written code, developers could take immediate corrective actions, improving the overall quality of the software. Additionally, automated decision-making systems could be developed to prioritize defect-prone modules for further testing or refactoring.

- **Addressing the Scalability of Predictive Models:** As software systems continue to grow in complexity, the scalability of predictive models becomes an important consideration. Future research could focus on developing models that can handle large-scale software systems with millions of lines of code and thousands of com-

ponents. Techniques such as distributed machine learning and parallel processing could be explored to improve the scalability of these models.

- **Incorporating More Real-World Datasets:** While this research has validated the proposed models using several datasets, future work could focus on collecting and utilizing more real-world datasets to ensure the robustness of the models. Collaborations with industry partners could provide access to proprietary datasets, enabling researchers to test their models in more diverse and realistic environments.

- **Enhancing Feature Selection and Dimensionality Reduction Techniques:** Feature selection and dimensionality reduction are critical steps in the development of predictive models. Future research could investigate the use of more advanced techniques, such as unsupervised learning or reinforcement learning, to improve the feature selection process. These techniques could help reduce the dimensionality of the data while retaining the most important information, leading to more efficient and accurate models.

- **Long-Term Maintenance of Predictive Models:** Software systems are constantly evolving, and the factors that contribute to software defects may change over time. Future research could focus on the long-term maintenance of predictive models, ensuring that they remain accurate and relevant as software systems evolve. Techniques such as model retraining, adaptive learning, and concept drift detection could be explored to ensure the continued effectiveness of the models over time.

This PhD research has made significant contributions to the field of software quality prediction by addressing key challenges such as imbalanced data, multi-collinearity, and

overfitting. The development of new machine learning techniques, combined with the exploration of advanced methodologies such as deep learning and swarm intelligence, has resulted in more accurate and reliable predictive models. These models have the potential to improve software quality, reduce maintenance costs, and enhance the efficiency of software development processes. While there are still several avenues for future work, the findings of this research provide a strong foundation for further exploration in this field.

Appendices

Descriptive Statistics of Datasets

The descriptive statistics of datasets used in this thesis as given below.

Table A1: Descriptive Statistics - Equinox (EQ) dataset of AEEEM Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
ck_oo_numberOfPrivateMethods	2.0833	5.8486	0	49
LDHH_lcom	0.0012	0.0031	0	0.0246
LDHH_fanIn	0.0016	0.0035	0	0.0234
numberOfNonTrivialBugsFoundUntil:	4.2994	7.9066	0	71
WCHU_numberOfPublicAttributes	0.2616	0.9492	0	7.74
WCHU_numberOfAttributes	0.9004	2.0179	0	16.25
CvsWEntropy	0.1543	0.3104	0	2.6708
LDHH_numberOfPublicMethods	0.0006	0.0016	0	0.0152
WCHU_fanIn	0.8878	1.8128	0	11.14
LDHH_numberOfPrivateAttributes	0.0009	0.0024	0	0.0196
CvsEntropy	5.4748	6.5463	0	38.2812
LDHH_numberOfPublicAttributes	0.0003	0.0016	0	0.0167
WCHU_numberOfPrivateMethods	0.5533	1.7659	0	16.3
WCHU_numberOfMethods	1.076	2.3461	0	16.32
ck_oo_numberOfPublicAttributes	1.4074	9.4986	0	137
ck_oo_noc	0.179	0.7247	0	6
numberOfCriticalBugsFoundUntil:	0.2191	0.6182	0	4
ck_oo_wmc	32.642	60.6795	0	534
LDHH_numberOfPrivateMethods	0.0006	0.0022	0	0.0247
WCHU_numberOfPrivateAttributes	0.6359	1.4816	0	11.3
CvsLogEntropy	0.1494	0.1801	0	1.1282
WCHU_noc	0.05	0.2717	0	2.06
LDHH_numberOfAttributesInherited	0.0004	0.0017	0	0.0141
WCHU_wmc	2.2434	4.7909	0	37.23
ck_oo_fanOut	7.1481	9.6269	0	67
ck_oo_numberOfLinesOfCode	122.0185	228.9282	0	1805
ck_oo_numberOfAttributesInherited	1.4506	6.5681	0	74
ck_oo_numberOfMethods	9.8704	12.7044	0	75
ck_oo_dit	1.2315	0.4903	1	3
ck_oo_fanIn	2.9537	4.7613	0	32
LDHH_noc	0	0.0005	0	0.0058
WCHU_dit	0.0531	0.2259	0	1.02
ck_oo_lcom	124.2284	355.9648	0	2775
WCHU_numberOfAttributesInherited	0.2324	0.7783	0	5.1
ck_oo_rfc	58.3364	114.8791	0	1009
LDHH_wmc	0.0031	0.0069	0	0.0474
LDHH_numberOfAttributes	0.0011	0.0031	0	0.0189
LDHH_numberOfLinesOfCode	0.0039	0.0083	0	0.0616
WCHU_fanOut	1.5568	2.8063	0	22.43
WCHU_lcom	1.6615	4.4146	0	32.49
ck_oo_cbo	9.6728	11.1615	0	77
WCHU_rfc	2.5196	5.4469	0	43.4

Table1 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
ck_oo_numberOfAttributes	6.7037	13.895	0	138
numberOfHighPriorityBugsFoundUntil:	0.0432	0.2912	0	4
ck_oo_numberOfPrivateAttributes	3.4969	6.8903	0	82
numberOfMajorBugsFoundUntil:	0.4815	1.2869	0	11
WCHU_numberOfPublicMethods	0.5939	1.2502	0	9.19
LDHH_dit	0	0.0001	0	0.0006
WCHU_cbo	2.0825	3.2408	0	22.41
CvsLinEntropy	0.0163	0.0214	0	0.149
WCHU_numberOfMethodsInherited	1.2094	1.8928	0	13.4
numberOfBugsFoundUntil:	4.5864	8.4529	0	78
LDHH_fanOut	0.0026	0.0051	0	0.0373
LDHH_numberOfMethodsInherited	0.0015	0.0024	0	0.0197
LDHH_rfc	0.0036	0.0078	0	0.0603
ck_oo_numberOfMethodsInherited	14.7222	14.1429	0	111
ck_oo_numberOfPublicMethods	5.7438	7.187	0	54
LDHH_cbo	0.0037	0.006	0	0.0396
WCHU_numberOfLinesOfCode	2.9297	6.2688	0	48.68
CvsExpEntropy	0.0342	0.0533	0	0.3287
LDHH_numberOfMethods	0.0014	0.0034	0	0.0278
bug	0.3981	0.4903	0	1

Table A2: Descriptive Statistics - Eclipse JDT Core (JDT) dataset of AEEEM Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
ck_oo_numberOfPrivateMethods	1.2979	5.2603	0	111
LDHH_lcom	0.0018	0.0044	0	0.0458
LDHH_fanIn	0.0028	0.0073	0	0.0802
numberOfNonTrivialBugsFoundUntil:	10.1494	19.4007	0	200
WCHU_numberOfPublicAttributes	0.3964	1.3209	0	15.2
WCHU_numberOfAttributes	0.7748	2.3172	0	32.47
CvsWEntropy	0.0533	0.1591	0	2.2853
LDHH_numberOfPublicMethods	0.0014	0.0039	0	0.0435
WCHU_fanIn	1.1258	2.8804	0	35.57
LDHH_numberOfPrivateAttributes	0.0006	0.0025	0	0.0379
CvsEntropy	10.6824	9.0234	0	49.9143
LDHH_numberOfPublicAttributes	0.0007	0.0025	0	0.0259
WCHU_numberOfPrivateMethods	0.3397	1.3624	0	27.64
WCHU_numberOfMethods	1.1983	2.6448	0	33.63
ck_oo_numberOfPublicAttributes	2.7452	13.676	0	312
ck_oo_noc	0.7121	2.1548	0	26
numberOfCriticalBugsFoundUntil:	0.4333	1.2185	0	15
ck_oo_wmc	58.3842	135.7227	0	1680
LDHH_numberOfPrivateMethods	0.0006	0.003	0	0.051
WCHU_numberOfPrivateAttributes	0.2847	0.8665	0	12.15
CvsLogEntropy	3.6678	4.0346	0	9.3076
WCHU_noc	0.0934	0.387	0	5.07

Table2 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
LDHH_numberOfAttributesInherited	0.0125	0.0176	0	0.0624
WCHU_wmc	3.158	6.1151	0	58.84
ck_oo_fanOut	7.3952	9.6769	0	93
ck_oo_numberOfLinesOfCode	224.7292	555.7005	0	7341
ck_oo_numberOfAttributesInherited	102.2197	148.6499	0	563
ck_oo_numberOfMethods	13.5998	23.636	0	403
ck_oo_dit	2.7272	1.7215	1	8
ck_oo_fanIn	5.3681	13.7216	0	137
LDHH_noc	0.0002	0.0014	0	0.0118
WCHU_dit	0.2514	0.4419	0	2.03
ck_oo_lcom	364.7272	3230.074	0	81003
WCHU_numberOfAttributesInherited	5.8824	8.8877	0	29.73
ck_oo_rfc	76.8746	180.9786	0	2603
LDHH_wmc	0.0066	0.014	0	0.1298
LDHH_numberOfAttributes	0.0016	0.0054	0	0.0678
LDHH_numberOfLinesOfCode	0.0073	0.0149	0	0.1284
WCHU_fanOut	1.2079	2.0565	0	15.32
WCHU_lcom	2.6139	15.9445	0	359.48
ck_oo_cbo	12.2166	17.8159	0	156
WCHU_rfc	3.3708	6.2749	0	56.95
ck_oo_numberOfAttributes	7.3862	21.9605	0	313
numberOfHighPriorityBugsFoundUntil:	0.4604	1.0716	0	10
ck_oo_numberOfPrivateAttributes	1.674	3.6271	0	40
numberOfMajorBugsFoundUntil:	1.1384	2.8642	0	38
WCHU_numberOfPublicMethods	0.7857	1.9944	0	30.53
LDHH_dit	0.0004	0.0009	0	0.0097
WCHU_cbo	2.0713	3.4922	0	35.59
CvsLinEntropy	0.0797	0.0781	0	0.2977
WCHU_numberOfMethodsInherited	3.1467	3.6572	0	25.5
numberOfBugsFoundUntil:	11.6399	22.1266	0	214
LDHH_fanOut	0.0026	0.0056	0	0.0492
LDHH_numberOfMethodsInherited	0.007	0.0097	0	0.0601
LDHH_rfc	0.0067	0.0135	0	0.1271
ck_oo_numberOfMethodsInherited	49.2367	50.3677	0	319
ck_oo_numberOfPublicMethods	8.9529	19.9674	0	387
LDHH_cbo	0.0052	0.0098	0	0.0904
WCHU_numberOfLinesOfCode	4.3068	8.6063	0	91.44
CvsExpEntropy	0.1206	0.1329	0	0.5802
LDHH_numberOfMethods	0.0023	0.0057	0	0.0598
bug	0.2066	0.4051	0	1

Table A3: Descriptive Statistics - Apache Lucene dataset of AEEEM Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
ck_oo_numberOfPrivateMethods	1.3271	4.3013	0	47
LDHH_lcom	0.0024	0.0068	0	0.0613
LDHH_fanIn	0.0076	0.0193	0	0.1536

Table3 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
numberOfNonTrivialBugsFoundUntil:	2.4805	4.9582	0	81
WCHU_numberOfPublicAttributes	0.1301	0.5405	0	7.14
WCHU_numberOfAttributes	0.4307	1.2795	0	15.39
CvsWEntropy	0.0768	0.2407	0	4.4902
LDHH_numberOfPublicMethods	0.0018	0.006	0	0.0478
WCHU_fanIn	1.3607	3.896	0	37.8
LDHH_numberOfPrivateAttributes	0.0013	0.0048	0	0.0427
CvsEntropy	4.0192	4.4177	0	39.6054
LDHH_numberOfPublicAttributes	0.0003	0.002	0	0.0236
WCHU_numberOfPrivateMethods	0.1755	0.9087	0	19.6
WCHU_numberOfMethods	0.5939	1.5044	0	22.08
ck_oo_numberOfPublicAttributes	1.1375	2.8016	0	27
ck_oo_noc	0.7221	2.9627	0	42
numberOfCriticalBugsFoundUntil:	0	0	0	0
ck_oo_wmc	23.6831	50.0644	0	484
LDHH_numberOfPrivateMethods	0.0005	0.0035	0	0.0561
WCHU_numberOfPrivateAttributes	0.3193	1.0414	0	13.4
CvsLogEntropy	0.2363	0.5753	0	4.4516
WCHU_noc	0.2229	1.009	0	12.22
LDHH_numberOfAttributesInherited	0.0009	0.0039	0	0.0291
WCHU_wmc	0.9359	2.2406	0	34.19
ck_oo_fanOut	4.1606	4.3197	0	28
ck_oo_numberOfLinesOfCode	105.9103	239.4338	0	2864
ck_oo_numberOfAttributesInherited	1.5282	5.2839	0	56
ck_oo_numberOfMethods	7.4153	8.8949	0	126
ck_oo_dit	1.7699	0.8634	1	5
ck_oo_fanIn	4.6671	13.6294	0	174
LDHH_noc	0.001	0.0057	0	0.0543
WCHU_dit	0.0673	0.2579	0	2.02
ck_oo_lcom	63.288	347.0276	0	7875
WCHU_numberOfAttributesInherited	0.1863	0.7414	0	7.09
ck_oo_rfc	33.3951	55.8531	0	908
LDHH_wmc	0.0044	0.0108	0	0.1002
LDHH_numberOfAttributes	0.0017	0.006	0	0.0635
LDHH_numberOfLinesOfCode	0.0063	0.0123	0	0.1081
WCHU_fanOut	0.8251	1.4235	0	13.32
WCHU_lcom	0.8755	4.4484	0	102.52
ck_oo_cbo	8.7352	14.6519	0	179
WCHU_rfc	1.1432	2.5501	0	41.16
ck_oo_numberOfAttributes	4.8538	7.2803	0	64
numberOfHighPriorityBugsFoundUntil:	0	0	0	0
ck_oo_numberOfPrivateAttributes	2.6194	4.931	0	45
numberOfMajorBugsFoundUntil:	0	0	0	0
WCHU_numberOfPublicMethods	0.4351	1.0765	0	13.34
LDHH_dit	0.0002	0.0011	0	0.0112
WCHU_cbo	2.0248	4.1038	0	37.87
CvsLinEntropy	0.0302	0.0344	0	0.2423
WCHU_numberOfMethodsInherited	2.6873	2.0534	0	14.71
numberOfBugsFoundUntil:	2.4805	4.9582	0	81
LDHH_fanOut	0.0039	0.0087	0	0.055

Table3 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
LDHH_numberOfMethodsInherited	0.011	0.0105	0	0.0601
LDHH_rfc	0.0059	0.012	0	0.1037
ck_oo_numberOfMethodsInherited	20.4414	14.5201	0	106
ck_oo_numberOfPublicMethods	4.78	5.3761	0	55
LDHH_cbo	0.0106	0.0211	0	0.1574
WCHU_numberOfLinesOfCode	1.3521	3.3419	0	60.56
CvsExpEntropy	0.0685	0.0803	0	0.5351
LDHH_numberOfMethods	0.0029	0.0083	0	0.0799
bug	0.0926	0.2901	0	1

Table A4: Descriptive Statistics - Mylyn (ML) dataset of AEEEM Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
ck_oo_numberOfPrivateMethods	1.0166	2.498	0	24
LDHH_lcom	0.0025	0.0085	0	0.0846
LDHH_fanIn	0.0072	0.0201	0	0.2397
numberOfNonTrivialBugsFoundUntil:	3.6552	5.9817	0	92
WCHU_numberOfPublicAttributes	0.038	0.2756	0	4.28
WCHU_numberOfAttributes	0.4697	1.2381	0	18.27
CvsWEntropy	0.0276	0.0981	0	1.9077
LDHH_numberOfPublicMethods	0.002	0.0069	0	0.0712
WCHU_fanIn	1.0071	2.6076	0	38.27
LDHH_numberOfPrivateAttributes	0.0021	0.007	0	0.083
CvsEntropy	5.2229	5.7918	0	38.7367
LDHH_numberOfPublicAttributes	0.0003	0.0031	0	0.066
WCHU_numberOfPrivateMethods	0.2075	0.7547	0	8.11
WCHU_numberOfMethods	0.5354	1.376	0	14.19
ck_oo_numberOfPublicAttributes	0.9323	6.9425	0	127
ck_oo_noc	0.4232	2.2089	0	49
numberOfCriticalBugsFoundUntil:	0.1262	0.4527	0	5
ck_oo_wmc	16.7836	30.4676	0	727
LDHH_numberOfPrivateMethods	0.0009	0.0041	0	0.0585
WCHU_numberOfPrivateAttributes	0.3852	1.1302	0	18.28
CvsLogEntropy	5.7878	4.4912	0	10.25
WCHU_noc	0.1138	0.5579	0	7.17
LDHH_numberOfAttributesInherited	0.0004	0.0022	0	0.0308
WCHU_wmc	0.9425	2.2086	0	28.26
ck_oo_fanOut	4.6391	6.2655	0	62
ck_oo_numberOfLinesOfCode	83.8357	228.2076	0	7509
ck_oo_numberOfAttributesInherited	0.6992	2.4843	0	25
ck_oo_numberOfMethods	7.811	9.6028	0	119
ck_oo_dit	1.457	0.7404	1	5
ck_oo_fanIn	3.6756	12.7915	0	223
LDHH_noc	0.0006	0.0033	0	0.0515
WCHU_dit	0.0717	0.3237	0	3.04
ck_oo_lcom	72.6821	300.1481	0	7021
WCHU_numberOfAttributesInherited	0.0971	0.4142	0	4.42

Table4 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
ck_oo_rfc	34.8217	67.284	0	1266
LDHH_wmc	0.0049	0.0141	0	0.1645
LDHH_numberOfAttributes	0.0029	0.0095	0	0.1094
LDHH_numberOfLinesOfCode	0.0077	0.0179	0	0.1678
WCHU_fanOut	1.301	2.2421	0	20.42
WCHU_lcom	0.7588	2.367	0	27.65
ck_oo_cbo	8.1643	15.2599	0	259
WCHU_rfc	1.3253	2.7968	0	32.61
ck_oo_numberOfAttributes	5.0188	10.1689	0	128
numberOfHighPriorityBugsFoundUntil:	4.2986	6.5444	0	85
ck_oo_numberOfPrivateAttributes	3.1649	5.9458	0	103
numberOfMajorBugsFoundUntil:	0.3179	0.8567	0	13
WCHU_numberOfPublicMethods	0.4073	1.1024	0	10.22
LDHH_dit	0.0002	0.0013	0	0.021
WCHU_cbo	2.0265	3.2913	0	39.3
CvsLinEntropy	0.1123	0.0837	0	0.3292
WCHU_numberOfMethodsInherited	1.2697	1.5263	0	14.21
numberOfBugsFoundUntil:	7.8287	11.5808	0	197
LDHH_fanOut	0.0072	0.0142	0	0.1367
LDHH_numberOfMethodsInherited	0.0078	0.0104	0	0.0899
LDHH_rfc	0.0064	0.0166	0	0.1794
ck_oo_numberOfMethodsInherited	14.5279	18.1737	0	226
ck_oo_numberOfPublicMethods	6.0381	7.8337	0	98
LDHH_cbo	0.0135	0.0253	0	0.2498
WCHU_numberOfLinesOfCode	1.7149	3.3876	0	35.59
CvsExpEntropy	0.1945	0.1569	0	0.7276
LDHH_numberOfMethods	0.0027	0.0089	0	0.0866
bug	0.1316	0.3381	0	1

Table A5: Descriptive Statistics - Eclipse PDE UI (PDE) dataset of AEEEM Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
ck_oo_numberOfPrivateMethods	2.2458	4.6974	0	52
LDHH_lcom	0.0016	0.0029	0	0.0231
LDHH_fanIn	0.002	0.0058	0	0.0856
numberOfNonTrivialBugsFoundUntil:	2.7996	5.5066	0	143
WCHU_numberOfPublicAttributes	0.1178	2.6475	0	94.18
WCHU_numberOfAttributes	0.7605	3.0191	0	94.18
CvsWEntropy	0.0416	0.0745	0	0.7823
LDHH_numberOfPublicMethods	0.0008	0.0018	0	0.0218
WCHU_fanIn	1.282	3.7789	0	62.58
LDHH_numberOfPrivateAttributes	0.0008	0.0023	0	0.0292
CvsEntropy	8.5437	6.7229	0	58.1518
LDHH_numberOfPublicAttributes	0.0002	0.0037	0	0.1037
WCHU_numberOfPrivateMethods	0.5112	1.2482	0	10.23
WCHU_numberOfMethods	1.0963	1.8695	0	14.26

Table5 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
ck_oo_numberOfPublicAttributes	1.8831	56.6045	0	2168
ck_oo_noc	0.5959	2.4342	0	46
numberOfCriticalBugsFoundUntil:	0.0668	0.3723	0	6
ck_oo_wmc	23.7488	31.4144	0	286
LDHH_numberOfPrivateMethods	0.0007	0.0019	0	0.0182
WCHU_numberOfPrivateAttributes	0.5491	1.1544	0	10.12
CvsLogEntropy	0.2696	0.4212	0	5.6466
WCHU_noc	0.1686	0.9956	0	14.29
LDHH_numberOfAttributesInherited	0.0008	0.0034	0	0.0426
WCHU_wmc	1.8758	3.1582	0	28.59
ck_oo_fanOut	6.676	7.1895	0	48
ck_oo_numberOfLinesOfCode	98.1643	128.6349	0	1326
ck_oo_numberOfAttributesInherited	3.9693	15.3287	0	206
ck_oo_numberOfMethods	9.6293	9.0173	0	82
ck_oo_dit	2.2806	1.565	1	9
ck_oo_fanIn	3.69	12.7823	0	355
LDHH_noc	0.0002	0.0014	0	0.0201
WCHU_dit	0.0777	0.3028	0	3.03
ck_oo_lcom	82.1757	210.8157	0	3321
WCHU_numberOfAttributesInherited	0.4524	1.4762	0	30.76
ck_oo_rfc	47.5023	63.1137	0	599
LDHH_wmc	0.0031	0.0058	0	0.065
LDHH_numberOfAttributes	0.0012	0.0063	0	0.1734
LDHH_numberOfLinesOfCode	0.0063	0.0079	0	0.0757
WCHU_fanOut	1.4777	2.2591	0	16.26
WCHU_lcom	1.4783	2.9981	0	30.81
ck_oo_cbo	10.2084	14.8314	0	362
WCHU_rfc	2.4335	3.7809	0	29.7
ck_oo_numberOfAttributes	5.4115	57.0468	0	2169
numberOfHighPriorityBugsFoundUntil:	0.0641	0.3673	0	6
ck_oo_numberOfPrivateAttributes	2.6754	3.8238	0	39
numberOfMajorBugsFoundUntil:	0.2418	0.7549	0	8
WCHU_numberOfPublicMethods	0.6381	1.283	0	11.37
LDHH_dit	0.0001	0.0004	0	0.004
WCHU_cbo	2.5055	4.3391	0	61.58
CvsLinEntropy	0.0266	0.0217	0	0.2876
WCHU_numberOfMethodsInherited	1.7793	2.4741	0	14.43
numberOfBugsFoundUntil:	3.8764	7.8857	0	232
LDHH_fanOut	0.0023	0.004	0	0.031
LDHH_numberOfMethodsInherited	0.0028	0.004	0	0.0209
LDHH_rfc	0.0036	0.0063	0	0.0678
ck_oo_numberOfMethodsInherited	28.6279	55.84	0	602
ck_oo_numberOfPublicMethods	5.5778	5.4219	0	53
LDHH_cbo	0.0042	0.0074	0	0.09
WCHU_numberOfLinesOfCode	3.2488	4.5713	0	34.77
CvsExpEntropy	0.0555	0.0468	0	0.5955
LDHH_numberOfMethods	0.0017	0.0032	0	0.0352
bug	0.1396	0.3467	0	1

Table A6: Descriptive Statistics - Apache ActiveMQ 5.0.0 dataset of JIRARepository

Metric	Mean	Standard Deviation	Minimum	Maximum
CountDeclMethodPrivate	0.2452	0.9941	0	21
AvgLineCode	6.6757	5.9576	0	63
CountLine	131.9804	149.5589	20	2075
MaxCyclomatic	2.9326	3.3871	0	56
CountDeclMethodDefault	0.6821	3.2163	0	44
AvgEssential	1.0637	0.3229	0	4
CountDeclClassVariable	0.664	1.5658	0	23
SumCyclomaticStrict	15.2091	23.9075	0	301
AvgCyclomatic	1.4931	0.9471	0	14
AvgLine	9.1688	7.8422	0	75
CountDeclClassMethod	0.2824	1.6226	0	45
AvgLineComment	1.5409	2.42	0	17
AvgCyclomaticModified	1.4894	0.9414	0	14
CountDeclFunction	9.1338	11.9956	0	156
CountLineComment	38.0398	42.204	16	1031
CountDeclClass	1.2542	1.0478	1	18
CountDeclMethod	9.1359	12.0006	0	156
SumCyclomaticModified	14.5674	22.4602	0	294
CountLineCodeDecl	27.7144	30.6632	2	367
CountDeclMethodProtected	1.4692	3.2252	0	40
CountDeclInstanceVariable	2.4904	4.9556	0	68
MaxCyclomaticStrict	3.1635	3.802	0	57
CountDeclMethodPublic	6.7394	10.417	0	129
CountLineCodeExe	40.5621	69.1249	0	929
SumCyclomatic	14.6354	22.6429	0	294
SumEssential	9.9618	13.6066	0	166
CountStmtDecl	28.3015	31.8711	2	394
CountLineCode	75.6826	106.268	3	1191
CountStmtExe	30.699	52.2825	0	593
RatioCommentToCode	1.1715	1.2044	0.03	13.17
CountLineBlank	18.6019	19.3011	1	299
CountStmt	59.0005	81.4542	2	987
MaxCyclomaticModified	2.9045	3.3182	0	56
CountSemicolon	44.0679	61.4349	1	891
AvgLineBlank	0.5414	1.2909	0	15
CountDeclInstanceMethod	8.8535	11.9034	0	155
AvgCyclomaticStrict	1.5441	1.0629	0	18
PercentLackOfCohesion	44.0409	39.2772	0	100
MaxInheritanceTree	2.44	1.595	0	8
CountClassDerived	0.8636	2.8387	0	52
CountClassCoupled	3.8503	5.9582	0	72
CountClassBase	1.3206	0.7267	0	8
CountInput_Max	6.5807	9.3636	0	142
CountInput_Mean	2.7701	2.8202	0	51.5
CountInput_Min	0.7787	2.0001	0	41
CountOutput_Max	7.5828	7.0639	0	50
CountOutput_Mean	3.5758	3.2718	0	38.8
CountOutput_Min	1.5308	2.5715	0	38
CountPath_Max	288.7075	6918.921	0	265302

Table6 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
CountPath_Mean	68.9014	1943.342	0	65536.5
CountPath_Min	1.0456	0.4845	0	15
MaxNesting_Max	1.1486	1.4289	0	8
MaxNesting_Mean	0.4473	0.569	0	4.2
MaxNesting_Min	0.0297	0.1987	0	2
COMM	2.8227	1.9509	1	23
ADEV	2.8227	1.9509	1	23
DDEV	1.2208	0.4493	1	4
Added_lines	44.5515	89.9829	1	1402
Del_lines	38.6953	80.0453	0	1326
OWN_LINE	0.7514	0.1958	0.3333	1
OWN_COMMIT	0.931	0.1447	0.3333	1
MINOR_COMMIT	0.0005	0.023	0	1
MINOR_LINE	2.0202	0.806	1	5
MAJOR_COMMIT	1.2203	0.4478	1	4
MAJOR_LINE	0.3217	0.588	0	4
RealBugCount	0.1555	0.3625	0	1

Table A7: Descriptive Statistics - Apache Derby 10.5.1 dataset of JIRARepository

Metric	Mean	Standard Deviation	Minimum	Maximum
CountDeclMethodPrivate	1.7527	5.9526	0	155
AvgLineCode	12.2495	25.3255	0	895
CountLine	377.1165	689.5627	23	12429
MaxCyclomatic	6.0817	9.0186	0	169
CountDeclMethodDefault	1.3627	4.3004	0	87
AvgEssential	1.1745	0.7007	0	11
CountDeclClassVariable	2.8063	13.4756	0	324
SumCyclomaticStrict	33.8932	70.7123	0	1285
AvgCyclomatic	2.1623	1.9782	0	23
AvgLine	20.3956	42.4236	0	1679
CountDeclClassMethod	1.6965	16.9658	0	854
AvgLineComment	6.0673	8.7308	0	275
AvgCyclomaticModified	2.0495	1.7101	0	21
CountDeclFunction	14.1294	27.5589	0	854
CountLineComment	130.0163	225.6699	15	3778
CountDeclClass	1.1863	0.8008	1	14
CountDeclMethod	14.1316	27.5579	0	854
SumCyclomaticModified	30.3423	61.1237	0	1014
CountLineCodeDecl	57.9933	99.4338	2	2345
CountDeclMethodProtected	1.0055	4.3691	0	89
CountDeclInstanceVariable	3.5227	23.9227	0	1179
MaxCyclomaticStrict	6.7664	10.4016	0	169
CountDeclMethodPublic	10.0107	23.7803	0	854
CountLineCodeExe	119.2983	304.8227	0	5142
SumCyclomatic	31.7749	64.778	0	1158
SumEssential	17.3627	32.8452	0	854

Table7 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
CountStmtDecl	46.234	74.6346	2	1379
CountLineCode	197.2865	416.8514	3	7360
CountStmtExe	75.1538	183.191	0	3222
RatioCommentToCode	1.5662	2.1831	0.05	24.83
CountLineBlank	53.4928	108.7038	2	3069
CountStmt	121.3878	246.5161	2	4127
MaxCyclomaticModified	5.4976	7.4497	0	79
CountSemicolon	92.353	195.0223	1	3593
AvgLineBlank	1.6747	11.2973	0	508
CountDeclInstanceMethod	12.4351	21.8793	0	310
AvgCyclomaticStrict	2.2954	2.1763	0	23
PercentLackOfCohesion	41.4259	38.6906	0	100
MaxInheritanceTree	2.064	1.4572	0	7
CountClassDerived	0.7974	5.6345	0	254
CountClassCoupled	8.0651	10.1345	0	133
CountClassBase	1.2396	0.7578	0	6
CountInput_Max	14.6503	51.3352	0	1476
CountInput_Mean	4.3601	9.4718	0	335
CountInput_Min	1.1409	6.9124	0	335
CountOutput_Max	9.9649	10.9118	0	103
CountOutput_Mean	3.7798	3.2952	0	52.5
CountOutput_Min	1.2314	1.5306	0	27
CountPath_Max	5526946	72264973	0	1E+09
CountPath_Mean	379311.2	6338737	0	1.67E+08
CountPath_Min	2.2078	50.2764	0	2600
MaxNesting_Max	1.7656	1.7099	0	8
MaxNesting_Mean	0.7362	0.758	0	5
MaxNesting_Min	0.0573	0.3161	0	5
COMM	0.2806	0.8647	0	20
ADEV	0.2806	0.8647	0	20
DDEV	0.2074	0.4674	0	5
Added_lines	18.7564	120.669	0	3487
Del_lines	1.9102	15.4245	0	407
OWN_LINE	0.8946	0.1393	0.2277	1
OWN_COMMIT	0.1801	0.3798	0	1
MINOR_COMMIT	0	0	0	0
MINOR_LINE	1.6218	0.8294	1	7
MAJOR_COMMIT	0.2074	0.4674	0	5
MAJOR_LINE	1.2118	1.7025	0	15
RealBugCount	0.1416	0.3487	0	1

Table A8: Descriptive Statistics - Apache HBase 0.94.0 dataset of JIRARepository

Metric	Mean	Standard Deviation	Minimum	Maximum
CountDeclMethodPrivate	2.0331	6.6756	0	165
AvgLineCode	9.8895	8.9047	0	97
CountLine	349.7592	1681.588	0	51285

Table8 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
MaxCyclomatic	5.1039	5.1873	0	45
CountDeclMethodDefault	1.4013	5.3019	0	82
AvgEssential	1.1596	0.5483	0	8
CountDeclClassVariable	3.6534	19.0895	0	560
SumCyclomaticStrict	45.882	354.0175	0	10982
AvgCyclomatic	1.9348	1.3182	0	13
AvgLine	13.5071	11.6428	0	137
CountDeclClassMethod	2.3022	9.5535	0	247
AvgLineComment	2.5779	3.2534	0	41
AvgCyclomaticModified	1.9178	1.3036	0	13
CountDeclFunction	20.2512	141.395	0	4355
CountLineComment	79.8867	179.6072	0	4519
CountDeclClass	2.7337	18.9085	0	582
CountDeclMethod	20.6091	146.6232	0	4519
SumCyclomaticModified	41.8111	309.8611	0	9599
CountLineCodeDecl	80.3853	307.9796	0	9242
CountDeclMethodProtected	0.644	3.2844	0	83
CountDeclInstanceVariable	5.7007	27.0896	0	810
MaxCyclomaticStrict	5.8839	7.0085	0	68
CountDeclMethodPublic	16.5307	135.2374	0	4189
CountLineCodeExe	137.6449	779.734	0	23721
SumCyclomatic	43.0425	329.2683	0	10214
SumEssential	26.6808	202.0728	0	6239
CountStmtDecl	68.8697	280.1913	0	8475
CountLineCode	232.8055	1330.344	0	40833
CountStmtExe	95.7356	632.9887	0	19462
RatioCommentToCode	2.0396	15.0585	0	344
CountLineBlank	39.288	219.0401	0	6720
CountStmt	164.6053	911.943	0	27937
MaxCyclomaticModified	5.0076	5.0094	0	45
CountSemicolon	120.2144	593.3091	0	17976
AvgLineBlank	0.5911	1.4878	0	15
CountDeclInstanceMethod	18.3069	138.3894	0	4272
AvgCyclomaticStrict	2.0378	1.425	0	13
PercentLackOfCohesion	58.9027	34.9464	0	100
MaxInheritanceTree	1.5732	0.8198	0	5
CountClassDerived	0.7828	2.8557	0	45
CountClassCoupled	7.7195	10.1679	0	132
CountClassBase	1.4419	0.8024	0	5
CountInput_Max	12.2852	30.2553	0	699
CountInput_Mean	4.0358	3.7974	0	45
CountInput_Min	1.1067	1.8683	0	20
CountOutput_Max	12.7753	11.1973	0	77
CountOutput_Mean	4.7842	3.9607	0	31
CountOutput_Min	1.5826	2.6032	0	31
CountPath_Max	243665	6860362	0	2.21E+08
CountPath_Mean	11212.86	356705.8	0	11606430
CountPath_Min	2.0992	23.6555	0	576
MaxNesting_Max	1.8905	1.6753	0	8
MaxNesting_Mean	0.7154	0.7212	0	6.4

Table8 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
MaxNesting_Min	0.0822	0.4077	0	6
COMM	3.6147	6.1672	0	87
ADEV	3.6147	6.1672	0	87
DDEV	1.8168	1.3081	0	9
Added_lines	150.6374	1307.651	0	41581
Del_lines	70.6308	645.435	0	20224
OWN_LINE	0.8368	0.1912	0	1
OWN_COMMIT	0.716	0.3038	0	1
MINOR_COMMIT	0.0415	0.3742	0	5
MINOR_LINE	1.865	1.1244	0	9
MAJOR_COMMIT	1.7753	1.1781	0	7
MAJOR_LINE	1.1303	1.5604	0	11
RealBugCount	0.2059	0.4045	0	1

Table A9: Descriptive Statistics - Apache Groovy 1.6 Beta 1 dataset of JIRA Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
CountDeclMethodPrivate	1.3106	10.3159	0	274
AvgLineCode	6.4762	6.8559	0	81
CountLine	172.1303	468.7334	6	10322
MaxCyclomatic	5.0207	16.2689	0	217
CountDeclMethodDefault	0.8027	8.0965	0	220
AvgEssential	1.1218	0.5154	0	8
CountDeclClassVariable	1.1596	8.4529	0	216
SumCyclomaticStrict	27.1681	79.5552	0	1248
AvgCyclomatic	1.7052	2.0244	0	28
AvgLine	8.0853	8.5697	0	91
CountDeclClassMethod	2.6784	29.814	0	596
AvgLineComment	1.0646	2.1991	0	23
AvgCyclomaticModified	1.5786	1.0639	0	10
CountDeclFunction	13.4629	37.4775	0	637
CountLineComment	47.0463	185.5383	0	5090
CountDeclClass	1.6797	2.6766	0	37
CountDeclMethod	13.4616	37.3955	0	633
SumCyclomaticModified	23.7686	70.6694	0	1205
CountLineCodeDecl	33.2473	73.2904	2	1222
CountDeclMethodProtected	1.0097	4.4746	0	97
CountDeclInstanceVariable	2.179	4.6372	0	73
MaxCyclomaticStrict	5.6724	17.5317	0	249
CountDeclMethodPublic	10.3386	30.1662	0	594
CountLineCodeExe	58.7527	182.6667	0	2718
SumCyclomatic	25.363	74.4387	0	1205
SumEssential	16.6699	46.0724	0	802
CountStmtDecl	34.2801	78.2202	2	1361
CountLineCode	102.5566	278.4355	2	4483
CountStmtExe	42.9233	132.6759	0	2040

Table9 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
RatioCommentToCode	1.2788	3.32	0	85.57
CountLineBlank	23.6784	51.5829	1	765
CountStmt	77.2034	208.985	2	3401
MaxCyclomaticModified	3.732	5.7187	0	97
CountSemicolon	53.5859	144.924	1	2238
AvgLineBlank	0.3337	1.213	0	13
CountDeclInstanceMethod	10.7832	21.8819	0	222
AvgCyclomaticStrict	1.8404	2.2453	0	28
PercentLackOfCohesion	36.8624	37.0927	0	100
MaxInheritanceTree	2.2314	1.3353	0	6
CountClassDerived	1.0244	3.9413	0	64
CountClassCoupled	4.1839	8.4525	0	96
CountClassBase	1.2412	0.6385	0	6
CountInput_Max	10.7004	27.6718	0	332
CountInput_Mean	3.188	5.5583	0	125.5
CountInput_Min	0.933	1.5466	0	24
CountOutput_Max	7.1023	10.3259	0	217
CountOutput_Mean	3.0476	2.4953	0	25
CountOutput_Min	1.4702	1.9855	0	25
CountPath_Max	1228197	34900962	0	1E+09
CountPath_Mean	2452.454	64062.06	0	1828155
CountPath_Min	1.1145	1.2529	0	28
MaxNesting_Max	1.3544	1.564	0	7
MaxNesting_Mean	0.5054	0.6293	0	4.25
MaxNesting_Min	0.0475	0.2686	0	3
COMM	1.9135	6.3169	0	139
ADEV	1.9135	6.3169	0	139
DDEV	0.6857	0.9895	0	8
Added_lines	63.6663	297.0319	0	5972
Del_lines	25.8928	129.7807	0	2225
OWN_LINE	0.7865	0.205	0.2627	1
OWN_COMMIT	0.3489	0.4319	0	1
MINOR_COMMIT	0.0097	0.1635	0	4
MINOR_LINE	2.1169	1.1406	1	7
MAJOR_COMMIT	0.676	0.9524	0	5
MAJOR_LINE	0.894	1.5632	0	21
bug	0.0853	0.2794	0	1

Table A10: Descriptive Statistics - Apache Hive 0.9.0 dataset of JIRA Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
CountDeclMethodPrivate	1.1081	8.2365	0	257
AvgLineCode	8.6794	10.2397	0	102
CountLine	287.1469	1901.059	1	63702
MaxCyclomatic	5.2542	8.0854	0	145
CountDeclMethodDefault	0.4484	3.6757	0	128
AvgEssential	1.2154	1.0369	0	21

Table10 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
CountDeclClassVariable	1.56	20.2047	0	752
SumCyclomaticStrict	45.4802	425.7567	0	15035
AvgCyclomatic	2.0586	1.8113	0	24
AvgLine	11.2239	12.6095	0	129
CountDeclClassMethod	1.1631	11.0083	0	385
AvgLineComment	1.6045	2.6757	0	23
AvgCyclomaticModified	1.935	1.5437	0	24
CountDeclFunction	18.1017	163.4339	0	5782
CountLineComment	48.2331	128.7335	0	3673
CountDeclClass	2.1434	13.1966	0	392
CountDeclMethod	14.8616	162.1528	0	6038
SumCyclomaticModified	40.3001	365.0319	0	12713
CountLineCodeDecl	60.1879	339.8931	0	11329
CountDeclMethodProtected	0.3446	3.5692	0	129
CountDeclInstanceVariable	5.3298	44.1741	0	1245
MaxCyclomaticStrict	5.9463	9.8648	0	152
CountDeclMethodPublic	12.9605	148.287	0	5524
CountLineCodeExe	121.0918	982.4837	0	31367
SumCyclomatic	42.803	393.1729	0	13753
SumEssential	26.0537	248.4058	0	8818
CountStmtDecl	53.6194	309.0671	0	10144
CountLineCode	202.9025	1560.205	1	52208
CountStmtExe	93.0699	813.8308	0	25426
RatioCommentToCode	1.2288	2.4583	0	43
CountLineBlank	38.0431	266.655	0	8876
CountStmt	146.9174	1101.378	1	35570
MaxCyclomaticModified	4.7564	6.7189	0	92
CountSemicolon	96.0247	712.512	0	22779
AvgLineBlank	0.6137	1.8417	0	35
CountDeclInstanceMethod	13.6984	151.8159	0	5653
AvgCyclomaticStrict	2.185	1.979	0	25
PercentLackOfCohesion	41.7931	38.4287	0	100
MaxInheritanceTree	1.6928	1.0538	0	5
CountClassDerived	0.8482	4.8932	0	104
CountClassCoupled	5.8842	13.1125	0	207
CountClassBase	1.4004	0.8176	0	5
CountInput_Max	8.4936	12.0663	0	128
CountInput_Mean	3.3539	3.806	0	37.75
CountInput_Min	1.0678	2.5886	0	32
CountOutput_Max	9.4047	12.2926	0	124
CountOutput_Mean	3.6372	3.7148	0	35
CountOutput_Min	1.5155	2.5	0	35
CountPath_Max	75903.05	1547563	0	51609602
CountPath_Mean	2007.176	50295.96	0	1844303
CountPath_Min	1.1066	1.259	0	24
MaxNesting_Max	1.5805	1.7458	0	9
MaxNesting_Mean	0.6677	0.7394	0	4
MaxNesting_Min	0.1017	0.4147	0	4
COMM	0.5374	1.5848	0	28
ADEV	0.5374	1.5848	0	28

Table10 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
DDEV	0.3814	0.7696	0	7
Added_lines	28.3178	245.5229	0	8331
Del_lines	14.0805	183.7571	0	6275
OWN_LINE	0.7862	0.225	0.2051	1
OWN_COMMIT	0.224	0.3936	0	1
MINOR_COMMIT	0.0014	0.0376	0	1
MINOR_LINE	2.0791	1.2186	1	7
MAJOR_COMMIT	0.3799	0.7611	0	7
MAJOR_LINE	0.7281	1.4793	0	13
bug	0.1999	0.4	0	1

Table A11: Descriptive Statistics - Apache Ruby 1.1.0 dataset of JIRA Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
CountDeclMethodPrivate	1.554	6.3818	0	92
AvgLineCode	6.2804	14.8724	0	337
CountLine	218.4993	456.7192	4	4228
MaxCyclomatic	6.8495	25.9094	0	410
CountDeclMethodDefault	0.6361	3.5599	0	50
AvgEssential	1.1395	0.8942	0	17
CountDeclClassVariable	1.9261	9.0263	0	133
SumCyclomaticStrict	34.5882	83.9593	0	780
AvgCyclomatic	1.6758	1.8786	0	26
AvgLine	7.6512	15.6744	0	337
CountDeclClassMethod	2.3981	8.4324	0	105
AvgLineComment	0.8126	1.8325	0	25
AvgCyclomaticModified	1.5417	1.2013	0	13
CountDeclFunction	14.9658	29.83	0	320
CountLineComment	45.5814	65.4465	0	745
CountDeclClass	1.9822	5.7775	0	136
CountDeclMethod	14.9808	29.851	0	320
SumCyclomaticModified	28.6744	65.0546	0	585
CountLineCodeDecl	44.2011	85.501	2	934
CountDeclMethodProtected	0.4624	2.1958	0	27
CountDeclInstanceVariable	3.1067	9.5951	0	120
MaxCyclomaticStrict	7.5636	28.4313	0	438
CountDeclMethodPublic	12.3283	25.2151	0	265
CountLineCodeExe	89.4063	269.7917	0	3684
SumCyclomatic	31.8413	75.4714	0	695
SumEssential	20.2079	43.5586	0	398
CountStmtDecl	41.2408	74.9594	2	662
CountLineCode	144.9959	352.8829	3	3710
CountStmtExe	65.5732	192.4923	0	2792
RatioCommentToCode	1.5548	2.2279	0	10.67
CountLineBlank	28.9412	77.8818	0	1399
CountStmt	106.814	252.2008	2	2799
MaxCyclomaticModified	4.6525	10.8705	0	174

Table11 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
CountSemicolon	79.4583	202.5887	1	2797
AvgLineBlank	0.2572	0.7546	0	8
CountDeclInstanceMethod	12.5828	27.531	0	312
AvgCyclomaticStrict	1.7839	2.139	0	29
PercentLackOfCohesion	39.3461	37.9462	0	100
MaxInheritanceTree	1.6416	0.9609	0	6
CountClassDerived	1.2955	5.9466	0	81
CountClassCoupled	7.4364	14.1304	0	146
CountClassBase	1.2804	0.7248	0	7
CountInput_Max	13.487	55.5068	0	1220
CountInput_Mean	3.329	3.6903	0	51
CountInput_Min	1.0465	2.009	0	36
CountOutput_Max	8.5308	13.1609	0	187
CountOutput_Mean	3.0604	2.6077	0	20
CountOutput_Min	1.1751	1.2538	0	20
CountPath_Max	2755230	52271747	0	1E+09
CountPath_Mean	33700.22	795239.7	0	21276598
CountPath_Min	1.041	2.0288	0	54
MaxNesting_Max	1.3133	1.7428	0	9
MaxNesting_Mean	0.4218	0.5573	0	3
MaxNesting_Min	0.026	0.1976	0	3
COMM	4.1587	9.3783	0	99
ADEV	4.1587	9.3783	0	99
DDEV	1.3488	1.6226	0	8
Added_lines	102.8208	382.4897	0	5928
Del_lines	60.1231	295.9409	0	5944
OWN_LINE	0.7872	0.2295	0.2614	1
OWN_COMMIT	0.4554	0.4173	0	1
MINOR_COMMIT	0.0643	0.3846	0	3
MINOR_LINE	2.1382	1.2975	1	7
MAJOR_COMMIT	1.2845	1.466	0	7
MAJOR_LINE	1.1943	1.6177	0	8
RealBugCount	0.119	0.324	0	1

Table A12: Descriptive Statistics - Apache Wicket 1.3.0 Beta 2 dataset of JIRA Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
CountDeclMethodPrivate	0.3301	1.4266	0	35
AvgLineCode	6.9665	6.7361	0	112
CountLine	138.6432	206.6088	24	3650
MaxCyclomatic	2.4135	2.9901	0	33
CountDeclMethodDefault	0.4532	1.9299	0	27
AvgEssential	1.0635	0.5632	0	17
CountDeclClassVariable	1.1027	2.4779	0	54
SumCyclomaticStrict	10.5428	22.1696	0	361
AvgCyclomatic	1.3488	1.0745	0	26

Table12 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
AvgLine	12.1872	8.9604	0	133
CountDeclClassMethod	0.2813	1.7084	0	34
AvgLineComment	4.4521	3.06	0	31
AvgCyclomaticModified	1.346	1.0678	0	26
CountDeclFunction	6.4373	10.7853	0	180
CountLineComment	62.7153	83.9143	16	1660
CountDeclClass	1.612	1.7361	1	23
CountDeclMethod	6.4362	10.7847	0	180
SumCyclomaticModified	9.9887	20.3141	0	332
CountLineCodeDecl	19.3823	26.0635	2	355
CountDeclMethodProtected	0.8366	2.2647	0	41
CountDeclInstanceVariable	1.198	2.989	0	53
MaxCyclomaticStrict	2.6353	3.6108	0	51
CountDeclMethodPublic	4.8162	8.3049	0	123
CountLineCodeExe	25.2167	50.8115	0	698
SumCyclomatic	10.0278	20.5395	0	332
SumEssential	7.2853	13.1842	0	206
CountStmtDecl	19.5581	26.5351	2	360
CountLineCode	62.1407	110.1732	3	1660
CountStmtExe	17.6126	37.186	0	507
RatioCommentToCode	1.9493	1.6508	0.17	15.6
CountLineBlank	13.9421	20.8007	1	334
CountStmt	37.1707	62.4186	2	867
MaxCyclomaticModified	2.4005	2.9087	0	33
CountSemicolon	26.637	43.1056	1	534
AvgLineBlank	0.3936	1.0463	0	12
CountDeclInstanceMethod	6.1548	10.4617	0	180
AvgCyclomaticStrict	1.4095	1.3632	0	36
PercentLackOfCohesion	51.9801	43.5395	0	100
MaxInheritanceTree	3.4986	2.2019	0	10
CountClassDerived	1.3579	8.3403	0	238
CountClassCoupled	3.7833	5.065	0	57
CountClassBase	1.3199	0.6736	0	12
CountInput_Max	4.9353	16.0631	0	544
CountInput_Mean	2.0096	2.7737	0	54.25
CountInput_Min	0.785	1.4626	0	28
CountOutput_Max	5.8338	5.5453	0	38
CountOutput_Mean	2.9622	2.2563	0	24
CountOutput_Min	1.5837	1.8324	0	24
CountPath_Max	1037.085	41375.67	0	1736592
CountPath_Mean	336.2706	13787.92	0	578868.3
CountPath_Min	6.0652	208.3565	0	8749
MaxNesting_Max	0.9257	1.4129	0	8
MaxNesting_Mean	0.347	0.5758	0	5
MaxNesting_Min	0.0318	0.2478	0	5
COMM	3.1004	2.8932	1	51
ADEV	3.1004	2.8932	1	51
DDEV	2.5576	1.1432	1	10
Added_lines	280.017	444.6721	25	8133
Del_lines	12.6716	58.9818	0	1022

Table12 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
OWN_LINE	0.6814	0.1987	0.2281	1
OWN_COMMIT	0.4917	0.1769	0.1538	1
MINOR_COMMIT	0.0199	0.2569	0	5
MINOR_LINE	2.8968	1.1746	1	6
MAJOR_COMMIT	2.5377	1.0734	1	8
MAJOR_LINE	1.0227	1.2122	0	8
bug	0.0737	0.2614	0	1

Table A13: Descriptive Statistics - Apache Ant 1.7 dataset of PROMISE Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
wmc	11.0711	11.976	0	120
dit	2.5221	1.3989	1	7
noc	0.7315	4.8004	0	102
cbo	11.047	26.3431	0	499
rfc	34.3624	36.025	0	288
lcom	89.1477	349.9376	0	6692
ca	5.655	25.8142	0	498
ce	5.7463	5.6532	0	37
npm	8.3651	9.3313	0	103
lcom3	1.0133	0.619	0	2
loc	280.0711	411.8721	0	4541
dam	0.6449	0.4381	0	1
moa	0.7262	1.4266	0	11
mfa	0.51	0.3987	0	1
cam	0.4747	0.2599	0	1
ic	0.7208	0.9389	0	5
cbm	1.3128	2.3326	0	19
amc	23.6409	76.9861	0	2052
max_cc	4.6698	6.2769	0	53
avg_cc	1.3661	0.8817	0	6.7778
defects	0.2228	0.4164	0	1

Table A14: Descriptive Statistics - Apache Camel 1.4 dataset of PROMISE Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
wmc	8.5229	10.7435	0	141
dit	1.9472	1.2829	0	6
noc	0.5218	2.5742	0	36
cbo	10.7626	20.8798	0	389
rfc	21.2007	23.7672	0	286
lcom	73.4174	429.9116	0	9792
ca	5.1078	20.0845	0	387
ce	6.3314	6.7926	0	69
npm	6.9174	9.696	0	130

Table14 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
lcom3	1.1055	0.7213	0	2
loc	112.4771	162.2365	0	1747
dam	0.6073	0.4786	0	1
moa	0.6044	1.1769	0	9
mfa	0.3925	0.4168	0	1
cam	0.4915	0.265	0	1
ic	0.3647	0.5771	0	3
cbm	0.6067	1.2478	0	15
amc	10.9107	11.6467	0	148.6667
max_cc	2.1583	2.6168	0	32
avg_cc	0.9434	0.6038	0	6.5
defects	0.1663	0.3725	0	1

Table A15: Descriptive Statistics - Apache Ivy 2.0 dataset of PROMISE Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
wmc	11.2841	15.1482	1	157
dit	1.7926	1.2448	1	6
noc	0.3693	1.3183	0	17
cbo	13.233	16.5711	1	150
rfc	34.0369	44.6796	1	312
lcom	131.5795	712.192	0	11794
ca	6.8807	13.9389	0	147
ce	5.1648	8.9313	0	75
npm	9.0369	12.6361	0	142
lcom3	1.0594	0.6601	0	2
loc	249.3438	428.2597	1	2894
dam	0.6162	0.4599	0	1
moa	0.7159	1.4417	0	12
mfa	0.2909	0.3852	0	1
cam	0.4908	0.2546	0.0552	1
ic	0.358	0.7336	0	4
cbm	0.6364	1.7811	0	18
amc	18.4897	27.0328	0	203.5
max_cc	3.1875	3.8481	0	29
avg_cc	1.2143	0.8161	0	6.5
defects	0.1136	0.3178	0	1

Table A16: Descriptive Statistics - jEdit 4.0 dataset of PROMISE Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
wmc	12.8824	30.9583	1	407
dit	2.7647	2.1187	1	8
noc	0.4412	2.6988	0	35
cbo	12.3954	18.0409	1	184

Table16 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
rfc	38.2418	57.0192	1	494
lcom	197.3758	1221.246	0	16336
ca	7.5131	15.7687	0	157
ce	6.4281	7.6267	0	59
npm	7.7026	16.2952	0	193
lcom3	1.0456	0.602	0	2
loc	473.2124	1584.691	1	23683
dam	0.5588	0.4647	0	1
moa	0.9346	1.9152	0	17
mfa	0.5353	0.4435	0	1
cam	0.4711	0.2492	0	1
ic	0.6732	0.9906	0	4
cbm	1.6144	3.2302	0	25
amc	31.3419	37.2195	0	496
max_cc	6.2255	9.433	0	84
avg_cc	1.7917	1.7975	0	20
defects	0.2451	0.4308	0	1

Table A17: Descriptive Statistics - Apache Log4j 1.0 dataset of PROMISE Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
wmc	6.5852	5.6336	0	49
dit	1.6	0.932	1	5
noc	0.3037	1.1081	0	9
cbo	6.8593	8.9893	0	60
rfc	21.4	17.5134	0	96
lcom	19.9333	65.4542	0	688
ca	3.6741	8.4003	0	53
ce	3.4519	3.1332	0	15
npm	4.4815	4.7108	0	42
lcom3	0.9969	0.6193	0	2
loc	159.6222	184.5988	0	1176
dam	0.1219	0.2758	0	1
moa	0.7926	1.2038	0	7
mfa	0.2933	0.3934	0	1
cam	0.4389	0.2232	0	1
ic	0.363	0.6302	0	2
cbm	0.7111	1.5105	0	10
amc	20.2124	17.7835	0	100.6
max_cc	3.3333	3.3099	0	23
avg_cc	1.3419	0.8729	0	6.25
defects	0.2519	0.4357	0	1

Table A18: Descriptive Statistics - Apache Poi 2.0 dataset of PROMISE Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
wmc	14.3025	13.8599	0	108
dit	1.7006	0.6685	1	5
noc	0.742	7.6676	0	125
cbo	8.6529	16.7644	0	156
rfc	29.6465	33.4041	0	341
lcom	103.7643	359.6833	0	4268
ca	4.5096	14.5024	0	154
ce	4.4777	8.9443	0	121
npm	12.1242	11.6518	0	93
lcom3	0.9675	0.5205	0	2
loc	296.7229	661.299	0	9849
dam	0.5103	0.3906	0	1
moa	0.9713	2.8737	0	34
mfa	0.3002	0.2814	0	1
cam	0.4231	0.1872	0	1
ic	0.5318	0.5186	0	2
cbm	2.6146	2.8252	0	7
amc	18.1292	38.7297	0	614.0625
max_cc	3.586	7.6368	0	117
avg_cc	1.1501	1.044	0	14.2222
defects	0.1178	0.3229	0	1

Table A19: Descriptive Statistics - Apache Tomcat dataset of PROMISE Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
wmc	12.9592	18.6189	0	252
dit	1.6876	1.053	1	6
noc	0.3636	1.9737	0	31
cbo	7.5734	11.0969	0	109
rfc	33.4709	44.9766	0	511
lcom	176.2762	1159.188	0	29258
ca	3.8625	8.9033	0	109
ce	0	0	0	0
npm	10.7762	16.7132	0	231
lcom3	1.0862	0.6604	0	2
loc	350.4359	644.839	0	7956
dam	0.5741	0.4714	0	1
moa	0.9441	2.1077	0	24
mfa	0.2938	0.3866	0	1
cam	0.4865	0.2536	0	1
ic	0.2751	0.5788	0	4
cbm	0.5932	1.7421	0	19
amc	25.5776	46.6422	0	894.5
max_cc	4.2716	6.9544	0	95
avg_cc	1.2505	1.0024	0	10
defects	0.0897	0.286	0	1

Table A20: Descriptive Statistics - Apache Velocity 1.6 dataset of PROMISE Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
wmc	9.0218	14.1351	0	153
dit	1.6769	0.8887	1	5
noc	0.4367	2.7548	0	39
cbo	10.8079	12.7227	0	80
rfc	22.9782	27.3691	0	250
lcom	80.3406	554.868	0	8092
ca	5.607	11.1797	0	76
ce	5.9825	7.676	0	61
npm	7.2183	8.7992	0	50
lcom3	1.2325	0.7107	0	2
loc	248.9607	1034.079	0	13175
dam	0.4321	0.4627	0	1
moa	0.4716	1.1453	0	10
mfa	0.3879	0.4115	0	1
cam	0.4651	0.2224	0	1
ic	0.3144	0.5516	0	2
cbm	0.4891	1.0413	0	9
amc	19.6087	28.1143	0	276
max_cc	3.9869	14.5893	0	209
avg_cc	1.2708	1.856	0	23
defects	0.3406	0.475	0	1

Table A21: Descriptive Statistics - Apache Xalan 2.4 dataset of PROMISE Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
wmc	11.4495	16.2951	0	123
dit	2.5657	1.5252	1	8
noc	0.6086	2.6089	0	29
cbo	14.4979	19.3436	0	171
rfc	30.1618	35.7323	0	355
lcom	130.0816	577.0398	0	6589
ca	6.7469	16.4663	0	155
ce	8.4606	10.003	0	64
npm	9.5519	13.9251	0	118
lcom3	1.1166	0.71	0	2
loc	311.325	513.4384	0	3479
dam	0.4478	0.4723	0	1
moa	0.9073	1.8737	0	18
mfa	0.5478	0.4401	0	1
cam	0.4658	0.2584	0	1
ic	0.9198	1.0859	0	5
cbm	3.0719	4.579	0	30
amc	26.8832	37.4275	0	436
max_cc	4.3762	6.4732	0	86
avg_cc	1.3466	1.2727	0	22

Table21 continued from previous page

Metric	Mean	Standard Deviation	Minimum	Maximum
defects	0.1521	0.3594	0	1

Table A22: Descriptive Statistics - Apache Xerces 1.3 dataset of PROMISE Repository

Metric	Mean	Standard Deviation	Minimum	Maximum
wmc	11.3775	13.3483	0	95
dit	2.0088	1.276	1	5
noc	0.4547	3.1392	0	52
cbo	5.0751	8.3892	0	60
rfc	21.7042	32.463	0	297
lcom	94.5232	250.1567	0	2425
ca	2.6667	6.8383	0	57
ce	2.755	4.7401	0	55
npm	8.8146	9.2035	0	55
lcom3	1.4652	0.657	0	2
loc	368.8631	1058.501	0	10701
dam	0.2646	0.4094	0	1
moa	0.8035	2.8031	0	41
mfa	0.3573	0.4249	0	1
cam	0.5037	0.2406	0	1
ic	0.3466	0.6595	0	4
cbm	1.3753	3.088	0	25
amc	21.964	58.8984	0	779.8
max_cc	3.4525	8.7705	0	147
avg_cc	1.236	1.1806	0	13.3585
defects	0.1523	0.3597	0	1

References

[1] I. G. Ndukwe, S. A. Licorish, A. Tahir, and S. G. MacDonell, "How have views on software quality differed over time? research and practice viewpoints," *Journal of Systems and Software*, vol. 195, p. 111524, 1 2023.

[2] B. R. Maxim and M. Kessentini, "An introduction to modern software quality assurance," *Software Quality Assurance: In Large Scale and Complex Software-intensive Systems*, pp. 19–46, 1 2016.

[3] M. Kuhn, "Building predictive models in r using the caret package," *Journal of Statistical Software*, vol. 28, 2008.

[4] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, pp. 707–740, 8 2016.

[5] "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[6] H. Krasner, "The cost of poor software quality in the us: A 2020 report," *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, vol. 2, 2021.

[7] J. Eloff, M. B. Bella, J. Eloff, and M. B. Bella, "Software failures: An overview," *Software Failure Investigation: A Near-Miss Analysis Approach*, pp. 7–24, 2018.

[8] J. Pachouly, S. Ahirrao, K. Kotecha, G. Selvachandran, and A. Abraham, "A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools," *Engineering Applications of Artificial Intelligence*, vol. 111, p. 104773, 5 2022.

[9] M. Kuhn, K. Johnson *et al.*, *Applied predictive modeling*. Springer, 2013, vol. 26.

[10] M. B. Bjerke and R. Renger, "Being smart about writing smart objectives," *Evaluation and Program Planning*, vol. 61, pp. 125–127, 4 2017.

[11] R. Malhotra and M. Cherukuri, "Software defect categorization based on maintenance effort and change impact using multinomial naïve bayes algorithm," in *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*. IEEE, 2020, pp. 1068–1073.

[12] G. Catolino, F. Palomba, F. A. Fontana, A. D. Lucia, A. Zaidman, and F. Ferrucci, "Improving change prediction models with code smell-related information," *Empirical Software Engineering*, vol. 25, pp. 49–95, 1 2020.

[13] G. Yenduri and T. R. Gadekallu, "A review on soft computing approaches for predicting maintainability of software: State-of-the-art, technical challenges, and future directions," *Expert Systems*, vol. 40, 8 2023.

[14] K. Sahu, F. A. Alzahrani, R. K. Srivastava, and R. Kumar, "Evaluating the impact

of prediction techniques: Software reliability perspective." *Computers, Materials Continua*, vol. 67, 2021.

[15] M. Fernandez-Diego, E. R. Mendez, F. Gonzalez-Ladron-De-Guevara, S. Abrahao, and E. Insfran, "An update on effort estimation in agile software development: A systematic literature review," *IEEE Access*, vol. 8, pp. 166 768–166 800, 2020.

[16] S. K. Pandey, R. B. Mishra, and A. K. Tripathi, "Machine learning based methods for software fault prediction: A survey," *Expert Systems with Applications*, vol. 172, p. 114595, 6 2021.

[17] G. Czibula, I. G. Chelaru, I. G. Czibula, and A. J. Molnar, "An unsupervised learning-based methodology for uncovering behavioural patterns for specific types of software defects," *Procedia Computer Science*, vol. 225, pp. 2644–2653, 1 2023.

[18] S. K. Pandey, R. B. Mishra, and A. K. Tripathi, "Bpdet: An effective software bug prediction model using deep representation and ensemble learning techniques," *Expert Systems with Applications*, vol. 144, p. 113085, 2020.

[19] G. Giray, K. E. Bennin, Ömer Köksal, Önder Babur, and B. Tekinerdogan, "On the use of deep learning in software defect prediction," *Journal of Systems and Software*, vol. 195, p. 111537, 1 2023.

[20] M. N. Uddin, B. Li, Z. Ali, P. Kefalas, I. Khan, and I. Zada, "Software defect prediction employing bilstm and bert-based semantic feature," *Soft Computing*, vol. 26, pp. 7877–7891, 8 2022.

[21] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[22] W. Li and S. Henry, “Object-oriented metrics that predict maintainability,” *Journal of systems and software*, vol. 23, no. 2, pp. 111–122, 1993.

[23] M. Lorenz and J. Kidd, *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.

[24] F. B. e Abreu and W. Melo, “Evaluating the impact of object-oriented design on software quality,” in *Software Metrics Symposium, 1996., Proceedings of the 3rd International*, 1996, pp. 90–99.

[25] J. M. Bieman and B.-K. Kang, “Cohesion and reuse in an object-oriented system,” *ACM SIGSOFT Software Engineering Notes*, vol. 20, pp. 259–262, 1995.

[26] L. C. Briand, J. W. Daly, and J. K. Wust, “A unified framework for coupling measurement in object-oriented systems,” *IEEE Transactions on software Engineering*, vol. 25, pp. 91–121, 1999.

[27] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. USA: Prentice Hall PTR, 2003.

[28] Y.-S. Lee, “Measuring the coupling and cohesion of an object-oriented program based on information flow,” in *Proc. Int. Conf. on Software Quality, 1995*, 1995, pp. 81–90.

[29] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[30] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics," in *Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403)*. IEEE, 1999, pp. 242–249.

[31] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2 2015.

[32] Z. Mahmood, D. Bowes, P. C. R. Lane, and T. Hall, "What is the impact of imbalance on software defect prediction performance?" in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2015, pp. 1–4.

[33] N. Japkowicz *et al.*, "Learning from imbalanced data sets: a comparison of various strategies," in *AAAI Workshop on Learning from Imbalanced Data Sets*, vol. 68, 2000, pp. 10–15.

[34] J. V. Hulse, T. M. Khoshgoftaar, and A. Napolitano, "Experimental perspectives on learning from imbalanced data," in *Proceedings of the 24th International Conference on Machine Learning*, 2007, pp. 935–942.

[35] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, pp. 1263–1284, 2009.

[36] T. M. Khoshgoftaar and K. Gao, "Feature selection with imbalanced data for software defect prediction," in *2009 International Conference on Machine Learning and Applications*, 2009, pp. 235–240.

[37] M. M. Ãzturk and A. Zengin, "Hsdd: a hybrid sampling strategy for class imbalance in defect prediction data sets," in *2016 Eleventh International Conference on Digital Information Management (ICDIM)*, 2016, pp. 225–234.

[38] L. Kumar and A. Sureka, "Feature selection techniques to counter class imbalance problem for aging related bug prediction: aging related bug prediction," in *Proceedings of the 11th Innovations in Software Engineering Conference*, 2018, pp. 1–11.

[39] A. Estabrooks, T. Jo, and N. Japkowicz, "A multiple resampling method for learning from imbalanced data sets," *Computational intelligence*, vol. 20, pp. 18–36, 2004.

[40] J. Ge, J. Liu, and W. Liu, "Comparative study on defect prediction algorithms of supervised learning software based on imbalanced classification data sets," in *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2018, pp. 399–406.

[41] O. Alan and P. D. C. Catal, "An outlier detection algorithm based on object-oriented metrics thresholds," in *2009 24th International Symposium on Computer and Information Sciences*, 2009, pp. 567–570.

[42] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to im-

prove bug prediction," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 600–604.

[43] L. Kumar and S. K. Rath, "Application of genetic algorithm as feature selection technique in development of effective fault prediction model," in *2016 IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics Engineering (UPCON)*, 2016, pp. 432–437.

[44] H. Lu, E. Kocaguneli, and B. Cukic, "Defect prediction between software versions with active learning and dimensionality reduction," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 312–322.

[45] F. AktaÅ and F. Buzluca, "A learning-based bug predicition method for object-oriented systems," in *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*, 2018, pp. 217–223.

[46] Z. Yang and H. Qian, "Automated parameter tuning of artificial neural networks for software defect prediction," in *Proceedings of the 2nd International Conference on Advances in Image Processing*, 2018, pp. 203–209.

[47] R. Malhotra, A. Rajpal, and D. Rathore, "Parameter tuning on software defect prediction using differential evolution simulated annealing," in *2018 International Conference on Big Data and Artificial Intelligence (BDAI)*, 2018, pp. 97–106.

[48] C. Cui, B. Liu, and G. Li, "A novel feature selection method for software fault prediction model," in *2019 Annual Reliability and Maintainability Symposium (RAMS)*, 2019, pp. 1–6.

[49] M. Cui, Y. Sun, Y. Lu, and Y. Jiang, "Study on the influence of the number of features on the performance of software defect prediction model," in *Proceedings of the 2019 3rd International Conference on Deep Learning Technologies*, 2019, pp. 32–37.

[50] B. A. Kitchenham, P. Rialette, B. David, O. P. Brereton, M. Turner, M. Niazi, and S. Linkman, "Systematic literature reviews in software engineering a tertiary study," *Information and Software Technology*, vol. 52, no. 8, pp. pp. 792–805, 2010.

[51] G. R. Marczyk, D. DeMatteo, and D. Festinger, *Essentials of research design and methodology*. John Wiley & Sons, 2010, vol. 2.

[52] J. S. Shirabad, T. J. Menzies *et al.*, "The promise repository of software engineering databases," *School of information technology and engineering, University of Ottawa, Canada*, vol. 24, no. 3, 2005.

[53] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 31–41.

[54] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn, "Mining software defects: Should we consider affected releases?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 654–665.

[55] "Android — github.com," https://github.com/android, [Accessed 30-07-2023].

[56] H. Snyder, "Literature review as a research methodology: An overview and guidelines," *Journal of Business Research*, vol. 104, pp. 333–339, 2019.

[57] C. J. Wu and M. S. Hamada, *Experiments: planning, analysis, and optimization*. John Wiley & Sons, 2011.

[58] R. E. Kirk, "Experimental design," *Sage Handbook of Quantitative Methods in Psychology*, pp. 23–45, 2009.

[59] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[60] S. S. Rathore and S. Kumar, "An empirical study of ensemble techniques for software fault prediction," *Applied Intelligence*, vol. 51, pp. 3615–3644, 2021.

[61] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *The Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[62] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794.

[63] A. J. Ferreira and M. A. Figueiredo, "Boosting algorithms: A review of methods, theory, and applications," *Ensemble Machine Learning: Methods and Applications*, pp. 35–85, 2012.

[64] A. Plaia, S. Buscemi, J. Fürnkranz, and E. L. Mencía, "Comparing boosting and bagging for decision trees of rankings," *Journal of Classification*, vol. 39, no. 1, pp. 78–99, 2022.

[65] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural networks for Perception*. Elsevier, 1992, pp. 65–93.

[66] Ö. F. Arar and K. Ayan, "Software defect prediction using cost-sensitive neural network," *Applied Soft Computing*, vol. 33, pp. 263–277, 2015.

[67] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: concepts, cnn architectures, challenges, applications, future directions," *Journal of Big Data*, vol. 8, pp. 1–74, 2021.

[68] C. MacNeill, "Apache Ant - Welcome — ant.apache.org," https://ant.apache.org, [Accessed 04-04-2023].

[69] "Apache Camel — camel.apache.org," https://camel.apache.org/, [Accessed 04-04-2023].

[70] "Apache Ivy — ant.apache.org," https://ant.apache.org/ivy/, [Accessed 04-04-2023].

[71] "jEdit - Programmer's Text Editor - overview — jedit.org," http://www.jedit.org, [Accessed 04-04-2023].

[72] "Apache Log4j — logging.apache.org," https://logging.apache.org/log4j, [Accessed 04-04-2023].

[73] "Apache Ant - Apache Props Antlib — ant.apache.org," https://ant.apache.org/antlibs/props/, [Accessed 04-04-2023].

[74] "Android Open Source Project — source.android.com," https://source.android.com/, [Accessed 30-07-2023].

[75] R. Malhotra, N. Pritam, K. Nagpal, and P. Upmanyu, "Defect collection and reporting system for git based open source software," in *2014 International Conference on Data Mining and Intelligent Computing (ICDMIC)*. IEEE, 2014, pp. 1–7.

[76] R. Malhotra, *Empirical research in software engineering: concepts, analysis, and applications*. CRC Press, 2016.

[77] R. Malhotra and M. Khanna, "A text mining framework for analyzing change impact and maintenance effort of software bug reports," *International Journal of Information Retrieval Research (IJIRR)*, vol. 12, no. 1, pp. 1–18, 2022.

[78] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 36, no. 2, pp. 111–133, 1974.

[79] J. Akosa, "Predictive accuracy: A misleading performance measure for highly imbalanced data," in *Proceedings of the SAS Global Forum*, vol. 12. SAS Institute Inc. Cary, NC, USA, 2017, pp. 1–4.

[80] S. Morasca and L. Lavazza, "On the assessment of software defect prediction models via roc curves," *Empirical Software Engineering*, vol. 25, pp. 3977–4019, 2020.

[81] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *The Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.

[82] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[83] B. Kitchenham, S. Charters *et al.*, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[84] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, pp. 375–407, 2010.

[85] C. Manjula and L. Florence, "Deep neural network based hybrid approach for software defect prediction using software metrics," *Cluster Computing*, vol. 22, no. Suppl 4, pp. 9847–9863, 2019.

[86] J. Zheng, "Cost-sensitive boosting neural networks for software defect prediction," *Expert Systems with Applications*, vol. 37, no. 6, pp. 4537–4543, 2010.

[87] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 683–711, 2018.

[88] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect-proneness prediction framework," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 356–370, 2010.

[89] H. Turabieh, M. Mafarja, and X. Li, "Iterated feature selection algorithms with layered recurrent neural network for software fault prediction," *Expert Systems with Applications*, vol. 122, pp. 27–42, 2019.

[90] I. Alsmadi and H. Najadat, "Evaluating the change of software fault behavior with dataset attributes based on categorical correlation," *Advances in Engineering Software*, vol. 42, no. 8, pp. 535–546, 2011.

[91] T. Shippey, D. Bowes, and T. Hall, "Automatically identifying code features for software defect prediction: Using ast n-grams," *Information and Software Technology*, vol. 106, pp. 142–160, 2019.

[92] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: an investigation on feature selection techniques," *Software: Practice and Experience*, vol. 41, no. 5, pp. 579–606, 2011.

[93] X. Cai, Y. Niu, S. Geng, J. Zhang, Z. Cui, J. Li, and J. Chen, "An under-sampled software defect prediction method based on hybrid multi-objective cuckoo search," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 5, p. e5478, 2020.

[94] K. Gao, T. M. Khoshgoftaar, and A. Napolitano, "A hybrid approach to coping with high dimensionality and class imbalance for software defect prediction," in *2012 11th International Conference on Machine Learning and Applications*, vol. 2. IEEE, 2012, pp. 281–288.

[95] P. R. Bal and S. Kumar, "Wr-elm: Weighted regularization extreme learning ma-

chine for imbalance learning in software fault prediction," *IEEE Transactions on Reliability*, vol. 69, no. 4, pp. 1355–1375, 2020.

[96] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 552–569, 2012.

[97] W. Rhmann, B. Pandey, G. Ansari, and D. K. Pandey, "Software fault prediction based on change metrics using hybrid algorithms: An empirical study," *Journal of King Saud University-Computer and Information Sciences*, vol. 32, no. 4, pp. 419–424, 2020.

[98] A. Okutan and O. T. Yıldız, "Software defect prediction using bayesian networks," *Empirical Software Engineering*, vol. 19, pp. 154–181, 2014.

[99] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[100] J. Deng, L. Lu, and S. Qiu, "Software defect prediction via lstm," *IET software*, vol. 14, no. 4, pp. 443–450, 2020.

[101] M. Liu, L. Miao, and D. Zhang, "Two-stage cost-sensitive learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 63, no. 2, pp. 676–686, 2014.

[102] H. Alsghaier and M. Akour, "Software fault prediction using particle swarm algorithm with genetic algorithm and support vector machine classifier," *Software: Practice and Experience*, vol. 50, no. 4, pp. 407–427, 2020.

[103] G. Czibula, Z. Marian, and I. G. Czibula, "Software defect prediction using relational association rule mining," *Information Sciences*, vol. 264, pp. 260–278, 2014.

[104] R. Yedida and T. Menzies, "On the value of oversampling for deep learning in software defect prediction," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 3103–3116, 2021.

[105] G. Abaei, A. Selamat, and H. Fujita, "An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction," *Knowledge-Based Systems*, vol. 74, pp. 28–39, 2015.

[106] A. Ali, N. Khan, M. Abu-Tair, J. Noppen, S. McClean, and I. McChesney, "Discriminating features-based cost-sensitive approach for software defect prediction," *Automated Software Engineering*, vol. 28, pp. 1–18, 2021.

[107] E. Erturk and E. A. Sezer, "A comparison of some soft computing methods for software fault prediction," *Expert systems with applications*, vol. 42, no. 4, pp. 1872–1879, 2015.

[108] M. Ulan, W. Löwe, M. Ericsson, and A. Wingkvist, "Weighted software metrics aggregation and its application to defect prediction," *Empirical Software Engineering*, vol. 26, no. 5, p. 86, 2021.

[109] K. Zhao, Z. Xu, M. Yan, T. Zhang, D. Yang, and W. Li, "A comprehensive investigation of the impact of feature selection techniques on crashing fault residence prediction models," *Information and Software Technology*, vol. 139, p. 106652, 2021.

[110] C. Jin and S.-W. Jin, "Prediction approach of software fault-proneness based on hybrid artificial neural network and quantum particle swarm optimization," *Applied Soft Computing*, vol. 35, pp. 717–725, 2015.

[111] P. Ardimento, L. Aversano, M. L. Bernardi, M. Cimitile, and M. Iammarino, "Just-in-time software defect prediction using deep temporal convolutional networks," *Neural Computing and Applications*, vol. 34, no. 5, pp. 3981–4001, 2022.

[112] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 17–26.

[113] M. Nevendra and P. Singh, "Empirical investigation of hyperparameter optimization for software defect count prediction," *Expert Systems with Applications*, vol. 191, p. 116217, 2022.

[114] S. N. das Dôres, L. Alves, D. D. Ruiz, and R. C. Barros, "A meta-learning framework for algorithm recommendation in software fault prediction," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 1486–1491.

[115] P. Manchala and M. Bisi, "Diversity based imbalance learning approach for software fault prediction using machine learning models," *Applied Soft Computing*, vol. 124, p. 109069, 2022.

[116] Z. Marian, I.-G. Mircea, I.-G. Czibula, and G. Czibula, "A novel approach for software defect prediction using fuzzy decision trees," in *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2016, pp. 240–247.

[117] Z. Wang, W. Tong, P. Li, G. Ye, H. Chen, X. Gong, and Z. Tang, "Bugpre: an intelligent software version-to-version bug prediction system using graph convolutional neural networks," *Complex & Intelligent Systems*, vol. 9, no. 4, pp. 3835–3855, 2023.

[118] A. Panichella, C. V. Alexandru, S. Panichella, A. Bacchelli, and H. C. Gall, "A search-based training algorithm for cost-aware defect prediction," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 2016, pp. 1077–1084.

[119] Z. Yang, L. Lu, and Q. Zou, "Ensemble kernel-mapping-based ranking support vector machine for software defect prediction," *IEEE Transactions on Reliability*, vol. 73, no. 1, pp. 664–679, 2023.

[120] S. S. Rathore and S. Kumar, "An empirical study of some software fault prediction techniques for the number of faults prediction," *Soft Computing*, vol. 21, pp. 7417–7434, 2017.

[121] M. Mafarja, T. Thaher, M. A. Al-Betar, J. Too, M. A. Awadallah, I. Abu Doush, and H. Turabieh, "Classification framework for faulty-software using enhanced exploratory whale optimizer-based feature selection scheme and random forest ensemble learning," *Applied Intelligence*, vol. 53, no. 15, pp. 18 715–18 757, 2023.

[122] T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, "Towards more accurate severity prediction and fixer recommendation of software bugs," *Journal of Systems and Software*, vol. 117, pp. 166–184, 2016.

[123] W. Ali, L. Bo, X. Sun, X. Wu, S. Memon, S. Siraj, and A. S. Ashton, "Automated software bug localization enabled by meta-heuristic-based convolutional neural

network and improved deep neural network," *Expert Systems with Applications*, vol. 232, p. 120562, 2023.

[124] S. Huda, S. Alyahya, M. M. Ali, S. Ahmad, J. Abawajy, H. Al-Dossari, and J. Yearwood, "A framework for software defect prediction and metric selection," *IEEE access*, vol. 6, pp. 2844–2858, 2017.

[125] X. Yu, H. Dai, L. Li, X. Gu, J. W. Keung, K. E. Bennin, F. Li, and J. Liu, "Finding the best learning to rank algorithms for effort-aware defect prediction," *Information and Software Technology*, vol. 157, p. 107165, 2023.

[126] C. Ni, W.-S. Liu, X. Chen, Q. Gu, D.-X. Chen, and Q.-G. Huang, "A cluster based feature selection method for cross-project software defect prediction," *Journal of Computer Science and Technology*, vol. 32, pp. 1090–1107, 2017.

[127] N. A. A. Khleel and K. Nehéz, "Software defect prediction using a bidirectional lstm network combined with oversampling techniques," *Cluster Computing*, vol. 27, no. 3, pp. 3615–3638, 2024.

[128] L. Chen, B. Fang, Z. Shang, and Y. Tang, "Tackling class overlap and imbalance problems in software defect prediction," *Software Quality Journal*, vol. 26, pp. 97–125, 2018.

[129] J. P. Meher, S. Biswas, and R. Mall, "Deep learning-based software bug classification," *Information and Software Technology*, vol. 166, p. 107350, 2024.

[130] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of

imbalanced learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1253–1269, 2018.

[131] A. M. Ismail, S. H. Ab Hamid, A. A. Sani, and N. N. M. Daud, "Toward reduction in false positives just-in-time software defect prediction using deep reinforcement learning," *IEEE Access*, 2024.

[132] G. Abaei, A. Selamat, and J. Al Dallal, "A fuzzy logic expert system to predict module fault proneness using unlabeled data," *Journal of King Saud University-Computer and Information Sciences*, vol. 32, no. 6, pp. 684–699, 2020.

[133] R. Garg and A. Bhargava, "Bug prediction based on deep neural network with reptile search optimization to enhance software reliability," *Multimedia Tools and Applications*, pp. 1–23, 2024.

[134] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, "Multi: Multi-objective effort-aware just-in-time software defect prediction," *Information and Software Technology*, vol. 93, pp. 1–13, 2018.

[135] H. Wang, B. Arasteh, K. Arasteh, F. S. Gharehchopogh, and A. Rouhi, "A software defect prediction method using binary gray wolf optimizer and machine learning algorithms," *Computers and Electrical Engineering*, vol. 118, p. 109336, 2024.

[136] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[137] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. USA: Elsevier Science Inc., 1977.

[138] J. Hernández-González, D. Rodriguez, I. Inza, R. Harrison, and J. A. Lozano, "Learning to classify software defects from crowds: a novel approach," *Applied Soft Computing*, vol. 62, pp. 579–591, 2018.

[139] R. Malhotra and M. Cherukuri, "A systematic review of hyperparameter tuning techniques for software quality prediction models," *Intelligent Data Analysis*, no. Preprint, pp. 1–19.

[140] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empirical Software Engineering*, vol. 20, pp. 1354–1383, 2015.

[141] M. Ummat, "Design and development of models for analyzing software evolution."

[142] H. Alsolai and M. Roper, "A systematic literature review of machine learning techniques for software maintainability prediction," *Information and Software Technology*, vol. 119, p. 106214, 2020.

[143] F. Matloob, T. M. Ghazal, N. Taleb, S. Aftab, M. Ahmad, M. A. Khan, S. Abbas, and T. R. Soomro, "Software defect prediction using ensemble learning: A systematic literature review," *IEEE Access*, vol. 9, pp. 98 754–98 771, 2021.

[144] R. Polikar, "Ensemble learning," *Ensemble machine learning: Methods and applications*, pp. 1–34, 2012.

[145] "Ibm spss statistics for windows."

[146] K. S. I. G. K. SIG), "Keras: Deep Learning for humans — keras.io," https://keras.io/, [Accessed 03-07-2023].

[147] Y. Qu, X. Chen, Y. Zhao, and X. Ju, "Impact of hyper parameter optimization for cross-project software defect prediction," *International Journal of Performability Engineering*, vol. 14, no. 6, p. 1291, 2018.

[148] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[149] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of Machine Learning Research*, vol. 13, no. 2, 2012.

[150] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Advances in Neural Information Processing Systems*, vol. 25, 2012.

[151] H. Alibrahim and S. A. Ludwig, "Hyperparameter optimization: Comparing genetic algorithm against grid search and bayesian optimization," in *2021 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2021, pp. 1551–1559.

[152] O. M. Maruf, "The impact of parameter optimization of ensemble learning on defect prediction," *Computer Science Journal of Moldova*, vol. 79, no. 1, pp. 85–128, 2019.

[153] Y. Yang, X. Xia, D. Lo, T. Bi, J. Grundy, and X. Yang, "Predictive models in software engineering: Challenges and opportunities," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–72, 2022.

[154] P. K. Kudjo, S. B. Aformaley, S. Mensah, and J. Chen, "The significant effect of parameter tuning on software vulnerability prediction models," in *2019 IEEE 19th*

International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE, 2019, pp. 526–527.

[155] A. L. Oliveira, P. L. Braga, R. M. Lima, and M. L. Cornélio, "Ga-based method for feature selection and parameters optimization for machine learning regression applied to software effort estimation," *information and Software Technology*, vol. 52, no. 11, pp. 1155–1166, 2010.

[156] S. Medapati, K.-I. Lin, and L. Sherrell, "Automated parameter estimation process for clustering algorithms used in software maintenance," in *Proceedings of the 51st ACM Southeast Conference*, 2013, pp. 1–6.

[157] L. Song, L. L. Minku, and X. Yao, "The impact of parameter tuning on software effort estimation using learning machines," in *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*, 2013, pp. 1–10.

[158] C. Jin and S.-W. Jin, "Software reliability prediction model based on support vector regression with improved estimation of distribution algorithms," *Applied Soft Computing*, vol. 15, pp. 113–120, 2014.

[159] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, pp. 594–623, 2013.

[160] W. Zhao, T. Tao, and E. Zio, "System reliability prediction by support vector regression with analytic selection and genetic algorithm parameters selection," *Applied Soft Computing*, vol. 30, pp. 792–802, 2015.

[161] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 321–332.

[162] R. Malhotra and A. Negi, "Reliability modeling using particle swarm optimization," *International Journal of System Assurance Engineering and Management*, vol. 4, pp. 275–283, 2013.

[163] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *Information and Software Technology*, vol. 76, pp. 135–146, 2016.

[164] Z.-Y. Ma, W. Zhang, J.-P. Wang, F.-S. Liu, K. Han, and F. Gao, "A military software reliability prediction model based on optimized svr algorithm," in *Proceedings of the 2019 3rd International Conference on Innovation in Artificial Intelligence*, 2019, pp. 258–262.

[165] W. Fu, V. Nair, and T. Menzies, "Why is differential evolution better than grid search for tuning defect predictors?" *arXiv preprint arXiv:1609.02613*, 2016.

[166] H. Osman, M. Ghafari, and O. Nierstrasz, "Hyperparameter optimization to improve bug prediction accuracy," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2017, pp. 33–38.

[167] L. Villalobos-Arias, C. Quesada-López, J. Guevara-Coto, A. Martínez, and M. Jenkins, "Evaluating hyper-parameter tuning using random search in support vector machines for software effort estimation," in *Proceedings of the 16th ACM Interna-*

tional Conference on Predictive Models and Data Analytics in Software Engineering, 2020, pp. 31–40.

[168] F. Khan, S. Kanwal, S. Alamri, and B. Mumtaz, "Hyper-parameter optimization of classifiers, using an artificial immune network and its application to software bug prediction," *IEEE Access*, vol. 8, pp. 20 954–20 964, 2020.

[169] A. Agrawal and T. Menzies, "Is" better data" better than" better data miners"? on the benefits of tuning smote for defect prediction," in *Proceedings of the 40th International Conference on Software engineering*, 2018, pp. 1050–1061.

[170] K. Lakra and A. Chug, "Improving software maintainability prediction using hyperparameter tuning of baseline machine learning algorithms," in *Applications of Artificial Intelligence and Machine Learning: Select Proceedings of ICAAAIML 2020*. Springer, 2021, pp. 679–692.

[171] M. Hosni, A. Idri, A. Abran, and A. B. Nassif, "On the value of parameter tuning in heterogeneous ensembles effort estimation," *Soft Computing*, vol. 22, pp. 5977–6010, 2018.

[172] T. Xia, R. Krishna, J. Chen, G. Mathew, X. Shen, and T. Menzies, "Hyperparameter optimization for effort estimation," *arXiv preprint arXiv:1805.00336*, 2018.

[173] K. Tameswar, G. Suddul, and K. Dookhitram, "A hybrid deep learning approach with genetic and coral reefs metaheuristics for enhanced defect detection in software," *International Journal of Information Management Data Insights*, vol. 2, no. 2, p. 100105, 2022.

[174] M. M. Öztürk, "Comparing hyperparameter optimization in cross-and within-project defect prediction: A case study," *Arabian Journal for Science and Engineering*, vol. 44, pp. 3515–3530, 2019.

[175] J. Lee, J. Choi, D. Ryu, and S. Kim, "Holistic parameter optimization for software defect prediction," *IEEE Access*, vol. 10, pp. 106 781–106 797, 2022.

[176] L. L. Minku, "A novel online supervised hyperparameter tuning procedure applied to cross-company software effort estimation," *Empirical Software Engineering*, vol. 24, no. 5, pp. 3153–3204, 2019.

[177] T. Labidi and Z. Sakhrawi, "On the value of parameter tuning in stacking ensemble model for software regression test effort estimation," *The Journal of Supercomputing*, vol. 79, no. 15, pp. 17 123–17 145, 2023.

[178] L.-N. Qin, "Software reliability prediction model based on pso and svm," in *2011 International Conference on Consumer Electronics, Communications and Networks (CECNet)*. IEEE, 2011, pp. 5236–5239.

[179] E. Frank, M. A. Hall, and I. H. Witten, *The WEKA workbench*. Morgan Kaufmann, 2016.

[180] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, "How bugs are born: a model to identify how bugs are introduced in software components," *Empirical Software Engineering*, vol. 25, pp. 1294–1340, 2020.

[181] D. D. Nucci, F. Palomba, R. Oliveto, and A. D. Lucia, "Dynamic selection of classifiers in bug prediction: An adaptive method," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, pp. 202–212, 6 2017.

[182] Q. Yang and X. Wu, "10 challenging problems in data mining research," *International Journal of Information Technology & Decision Making*, vol. 5, no. 04, pp. 597–604, 2006.

[183] D. A. Cieslak and N. V. Chawla, "Start globally, optimize locally, predict globally: Improving performance on imbalanced data," in *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 2008, pp. 143–152.

[184] E. A. Felix and S. P. Lee, "Systematic literature review of preprocessing techniques for imbalanced data," *IET Software*, vol. 13, no. 6, pp. 479–496, 2019.

[185] G. E. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *ACM SIGKDD Explorations Newsletter*, vol. 6, no. 1, pp. 20–29, 2004.

[186] H. Kaur, H. S. Pannu, and A. K. Malhi, "A systematic review on imbalanced data challenges in machine learning: Applications and solutions," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–36, 2019.

[187] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[188] H. Han, W.-Y. Wang, and B.-H. Mao, "Borderline-smote: a new over-sampling method in imbalanced data sets learning," in *International Conference on Intelligent Computing*. Springer, 2005, pp. 878–887.

[189] H. He, Y. Bai, E. A. Garcia, and S. Li, "Adasyn: Adaptive synthetic sampling approach for imbalanced learning," in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*. Ieee, 2008, pp. 1322–1328.

[190] G. Douzas, F. Bacao, and F. Last, "Improving imbalanced learning through a heuristic oversampling method based on k-means and smote," *Information Sciences*, vol. 465, pp. 1–20, 2018.

[191] S.-J. Yen and Y.-S. Lee, "Cluster-based under-sampling approaches for imbalanced data distributions," *Expert Systems with Applications*, vol. 36, no. 3, pp. 5718–5727, 2009.

[192] A. Tanimoto, S. Yamada, T. Takenouchi, M. Sugiyama, and H. Kashima, "Improving imbalanced classification using near-miss instances," *Expert Systems with Applications*, vol. 201, p. 117130, 2022.

[193] P. Hart, "The condensed nearest neighbor rule (corresp.)," *IEEE Transactions on Information Theory*, vol. 14, no. 3, pp. 515–516, 1968.

[194] "Two modifications of cnn," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-6, no. 11, pp. 769–772, 1976.

[195] D. L. Wilson, “Asymptotic properties of nearest neighbor rules using edited data,” *IEEE Transactions on Systems, Man, and Cybernetics*, no. 3, pp. 408–421, 1972.

[196] G. E. Batista, A. L. Bazzan, M. C. Monard *et al.*, “Balancing training data for automated annotation of keywords: a case study.” *Wob*, vol. 3, pp. 10–18, 2003.

[197] G. LemaÃŽtre, F. Nogueira, and C. K. Aridas, “Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning,” *Journal of Machine Learning Research*, vol. 18, no. 17, pp. 1–5, 2017.

[198] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “An empirical comparison of model validation techniques for defect prediction models,” *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2016.

[199] S. Feng, J. Keung, X. Yu, Y. Xiao, and M. Zhang, “Investigation on the stability of smote-based oversampling techniques in software defect prediction,” *Information and Software Technology*, vol. 139, p. 106662, 2021.

[200] Q. Wang, Y. Ma, K. Zhao, and Y. Tian, “A comprehensive survey of loss functions in machine learning,” *Annals of Data Science*, pp. 1–26, 2020.

[201] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[202] R. Malhotra and M. Cherukuri, “Multi-collinearity in software quality prediction: Review of challenges and solutions,” *International Journal of Management, Technology And Engineering*, vol. 14, no. 8, p. 349â363, Aug 2024.

[203] S. Stradowski and L. Madeyski, "Machine learning in software defect prediction: A business-driven systematic mapping study," p. 107128, 2023.

[204] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang, "Perceptions, expectations, and challenges in defect prediction," *IEEE Transactions on Software Engineering*, vol. 46, no. 11, pp. 1241–1266, 2018.

[205] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Information and Software Technology*, vol. 58, pp. 388–402, 2015.

[206] P. Dhal and C. Azad, "A comprehensive survey on feature selection in the various fields of machine learning," *Applied Intelligence*, vol. 52, no. 4, pp. 4543–4581, 2022.

[207] M. Rostami, K. Berahmand, E. Nasiri, and S. Forouzandeh, "Review of swarm intelligence-based feature selection methods," *Engineering Applications of Artificial Intelligence*, vol. 100, p. 104210, 2021.

[208] J. Kennedy, "Swarm intelligence," in *Handbook of nature-inspired and innovative computing: integrating classical models with emerging technologies*. Springer, 2006, pp. 187–219.

[209] M. Ghosh, R. Guha, R. Sarkar, and A. Abraham, "A wrapper-filter feature selection technique based on ant colony optimization," *Neural Computing and Applications*, vol. 32, pp. 7839–7857, 2020.

[210] D. Rodrigues, L. A. Pereira, T. Almeida, J. P. Papa, A. Souza, C. C. Ramos, and X.-S. Yang, "Bcs: A binary cuckoo search algorithm for feature selection," in *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2013, pp. 465–468.

[211] B. Samieiyan, P. MohammadiNasab, M. A. Mollaei, F. Hajizadeh, and M. Kangavari, "Novel optimized crow search algorithm for feature selection," *Expert Systems with Applications*, vol. 204, p. 117486, 2022.

[212] T. Sharma and D. Spinellis, "Do we need improved code quality metrics?" *arXiv preprint arXiv:2012.12324*, 2020.

[213] D. E. Farrar and R. R. Glauber, "Multicollinearity in regression analysis: the problem revisited," *The Review of Economic and Statistics*, pp. 92–107, 1967.

[214] G. C. Wang, "How to handle multicollinearity in regression modeling," *The Journal of Business Forecasting*, vol. 15, no. 1, p. 23, 1996.

[215] N. E. Fenton and M. Neil, "Software metrics: successes, failures and new directions," *Journal of Systems and Software*, vol. 47, no. 2-3, pp. 149–157, 1999.

[216] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The impact of correlated metrics on defect models," *arXiv preprint arXiv:1801.10271*, 2018.

[217] L. Madeyski and M. Jureczko, "Which process metrics can significantly improve defect prediction models? an empirical study," *Software Quality Journal*, vol. 23, pp. 393–422, 2015.

[218] D. Chicco, L. Oneto, and E. Tavazzi, "Eleven quick tips for data cleaning and feature engineering," *PLOS Computational Biology*, vol. 18, no. 12, p. e1010718, 2022.

[219] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, 1996.

[220] Y.-F. Li, M. Xie, and T.-N. Goh, "Adaptive ridge regression system for software cost estimating on multi-collinear datasets," *Journal of Systems and Software*, vol. 83, no. 11, pp. 2332–2343, 2010.

[221] X. Yang and W. Wen, "Ridge and lasso regression models for cross-version defect prediction," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 885–896, 2018.

[222] M. H. Ahmad, R. Adnan, and N. Adnan, "A comparative study on some methods for handling multicollinearity problems," *Matematika*, pp. 109–119, 2006.

[223] A. Garg and K. Tai, "Comparison of statistical and machine learning methods in modelling of data with multicollinearity," *International Journal of Modelling, Identification and Control*, vol. 18, no. 4, pp. 295–312, 2013.

[224] M. Alibuhtto and T. Peiris, "Principal component regression for solving multi-collinearity problem," 2015.

[225] A. Katrutsa and V. Strijov, "Comprehensive study of feature selection methods to solve multicollinearity problem according to evaluation criteria," *Expert Systems with Applications*, vol. 76, pp. 1–11, 2017.

[226] J. I. Daoud, "Multicollinearity and regression analysis," in *Journal of Physics: Conference Series*, vol. 949, no. 1. IOP Publishing, 2017, p. 012009.

[227] D. Weaving, B. Jones, M. Ireton, S. Whitehead, K. Till, and C. B. Beggs, "Overcoming the problem of multicollinearity in sports performance data: A novel application of partial least squares correlation analysis," *PLoS One*, vol. 14, no. 2, p. e0211776, 2019.

[228] C. Tırınk, S. H. Abacı, and H. Onder, "Comparison of ridge regression and least squares methods in the presence of multicollinearity for body measurements in saanen kids," *Journal of the Institute of Science and Technology*, vol. 10, no. 2, pp. 1429–1437, 2020.

[229] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman, "Developer-related factors in change prediction: an empirical assessment," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 186–195.

[230] F. Drobnič, A. Kos, and M. Pustišek, "On the interpretability of machine learning models and experimental feature selection in case of multicollinear data," *Electronics*, vol. 9, no. 5, p. 761, 2020.

[231] N. Bettenburg and A. E. Hassan, "Studying the impact of social interactions on software quality," *Empirical Software Engineering*, vol. 18, pp. 375–431, 2013.

[232] J. C. Westland, "The cost of errors in software development: evidence from industry," *Journal of Systems and Software*, vol. 62, no. 1, pp. 1–9, 2002.

[233] R. M. Oâbrien, "A caution regarding rules of thumb for variance inflation factors," *Quality & Quantity*, vol. 41, pp. 673–690, 2007.

[234] T. Kyriazos and M. Poga, "Dealing with multicollinearity in factor analysis: the problem, detections, and solutions," *Open Journal of Statistics*, vol. 13, no. 3, pp. 404–424, 2023.

[235] W. Yoo, R. Mayberry, S. Bae, K. Singh, Q. P. He, and J. W. Lillard Jr, "A study of effects of multicollinearity in the multivariable analysis," *International Journal of Applied Science and Technology*, vol. 4, no. 5, p. 9, 2014.

[236] H. Midi, S. K. Sarkar, and S. Rana, "Collinearity diagnostics of binary logistic regression model," *Journal of Interdisciplinary Mathematics*, vol. 13, no. 3, pp. 253–267, 2010.

[237] I. S. Dar, S. Chand, M. Shabbir, and B. G. Kibria, "Condition-index based new ridge regression estimator for linear regression model with multicollinearity," *Kuwait Journal of Science*, vol. 50, no. 2, pp. 91–96, 2023.

[238] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[239] D. N. Schreiber-Gregory, "Ridge regression and multicollinearity: An in-depth review," *Model Assisted Statistics and Applications*, vol. 13, no. 4, pp. 359–365, 2018.

[240] R. M. Bastiaan, D. T. Salaki, and D. Hatidja, "Comparing the performance of

prediction model of ridge and elastic net in correlated dataset," in *Operations Research: International Conference Series*, vol. 3, no. 1, 2022, pp. 8–13.

[241] A. Lukkarinen, L. Malmi, and L. Haaranen, "Event-driven programming in programming education: a mapping review," *ACM Transactions on Computing Education (TOCE)*, vol. 21, no. 1, pp. 1–31, 2021.

[242] J. Paykin, N. R. Krishnaswami, and S. Zdancewic, "The essence of event-driven programming," *Leibniz, Leibniz International Proceedings in Informatics*, 2016.

[243] G. Chadha, S. Mahlke, and S. Narayanasamy, "Efetch: optimizing instruction fetch for event-driven webapplications," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, 2014, pp. 75–86.

[244] X. Fan, O. Sinnen, and N. Giacaman, "Balancing parallelization and asynchronization in event-driven programs with openmp," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 4, p. e4959, 2019.

[245] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazieres, and R. Morris, "Event-driven programming for robust software," in *Proceedings of the 10th workshop on ACM SIGOPS European Workshop*, 2002, pp. 186–189.

[246] K. D. Lee and K. D. Lee, "Event-driven programming," *Python Programming Fundamentals*, pp. 149–165, 2011.

[247] G. C. Philip, "Software design guidelines for event-driven programming," *Journal of Systems and Software*, vol. 41, no. 2, pp. 79–91, 1998.

[248] J. Fischer, R. Majumdar, and T. Millstein, "Tasks: language support for event-driven programming," in *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 2007, pp. 134–143.

[249] S. Tilkov and S. Vinoski, "Node. js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.

[250] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial use of metrics for object-oriented software: An exploratory analysis," *IEEE Transactions on software Engineering*, vol. 24, no. 8, pp. 629–639, 1998.

[251] B. Henderson-Sellers, "The mathematical validity of software metrics," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 5, pp. 89–94, 1996.

[252] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[253] J. Palsberg, "Keynote: Event driven software quality," in *2009 International Conference on Embedded Software and Systems*. IEEE, 2009, pp. xxi–xxi.

[254] H. Shah and T. R. Soomro, "Node. js challenges in implementation," *Global Journal of Computer Science and Technology*, vol. 17, no. 2, pp. 73–83, 2017.

[255] P. Ganty and R. Majumdar, "Analyzing real-time event-driven programs," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2009, pp. 164–178.

[256] M. Chmiel and E. Hrynkiewicz, "An idea of event-driven program tasks execution," *IFAC Proceedings Volumes*, vol. 42, no. 1, pp. 17–22, 2009.

[257] H. Hosobe, “Testing event-driven programs in processing,” in *Proceedings of the 2020 European Symposium on Software Engineering*, 2020, pp. 6–11.

Supervisor's Biography

Prof. Ruchika Malhotra
Professor & HoD
Department of Software Engineering
Delhi Technological University
Email: ruchikamalhotra@dtu.ac.in

Educational Qualifications:
Postdoc (Indiana University-Purdue University Indianapolis, USA), Ph.D (Computer Applications)

Prof. Ruchika Malhotra is Head of Department and Professor in the Department of Software Engineering, Delhi Technological University, Delhi, India. She served as Associate Dean in Industrial Research and Development, Delhi Technological University from August 2018 to 2022. She was awarded with prestigious Raman Fellowship for pursuing Postdoctoral research in Indiana University Purdue University Indianapolis USA. She received her master's and doctorate degree in software engineering from the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. She has received IBM Faculty Award 2013. She has been ranked amongst the World's top 2% scientist by Stanford University report, USA, for her research in the field of 'Artificial Intelligence and Image Processing' in 2020, 2021, 2022, and 2023. She is recipient of Commendable Research Award (in 2018, 2019, 2020, 2021, 2022, and 2023) by Delhi Technological University. Her H-index is 37 as reported by Google Scholar. She is author of book titled 'Empirical Research in Software Engineering' published by CRC press and co-author of a book on Object-Oriented Software Engineering published by PHI Learning. She has published more than 245 research papers in international journals and conferences. Her research interests are in software testing, improving software quality, statistical and adaptive prediction models, software metrics and the definition and validation of software metrics.

Author's Biography

Madhukar Cherukuri
EDP Manager and Research Scholar
Department of Software Engineering
Delhi Technological University
Email: madhukar@dtu.ac.in

Educational Qualifications:
B.Tech. (CSE)

Madhukar Cherukuri is currently pursuing his doctoral degree from Delhi Technological University, where he also holds the position of EDP Manager. He earned his bachelor's degree in Computer Science Engineering in 2010 from Andhra University, Visakhapatnam, Andhra Pradesh, India. He possesses over 14 years of professional experience across the IT industry, research, academia, and administration. He holds technical certifications from Oracle and IBM. He has been recognized with the Outstanding Performer Award from USAA for his exemplary contributions. Madhukar has authored and co-authored several research papers published in renowned international conferences and journals. Additionally, he has co-authored the book "Deep Learning and its Applications". He has also filed multiple patents. His research interests span across artificial intelligence, machine learning, software engineering, predictive modeling, data analytics, and cyber security.