DEVELOPMENT AND VALIDATION OF FEATURE SELECTION TECHNIQUES FOR SOFTWARE DEFECT PREDICTION

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

DOCTOR OF PHILOSOPHY
by

KISHWAR KHAN
Roll No.2K19/PHDCO/02

Under the Supervision of
PROF. RUCHIKA MALHOTRA
Professor and 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

December, 2024

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

CANDIDATE'S DECLARATION

I, **Kishwar Khan (2K19/PHDCO/02)**, hereby declare that the work which is being presented in the thesis entitled **"Development and Validation of Feature Selection Technique for Software Defect Prediction"** in the 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 authentic record of my own work carried out during the period from **2019** 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.

Date :

Place : New Delhi

Kishwar Khan
kishwarkhan037@gmail.com
Roll No.: 2K19/PHDCO/02
Department Of Software Engineering,
Delhi Technological University (DTU),
New Delhi -110042

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Main Bawana Road, Delhi-42

CERTIFICATE BY THE SUPERVISOR

Certified that **Kishwar Khan (2K19/PHDCO/02)** has carried out her research work presented in this thesis entitled **"Development and Validation of Feature Selection Technique for Software Defect Prediction"** for the award of **Doctor of Philosophy** from the 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 of the award of any other degree to the candidate or to anybody else from this or any other institution.

Supervisor

Prof. RUCHIKA MALHOTRA

Professor and Head of Department

Department of Software Engineering

Delhi Technological University, Delhi 110042

Date:

”This thesis is lovingly dedicated to my grandparents ***Mrs. Alimun Nisa*** *and* ***Mr. Shamsul Haq Khan****, whose unwavering support, wisdom, and encouragement have been a constant source of inspiration throughout my journey.”*

Acknowledgment

First and foremost, I would like to express my deepest gratitude to the Almighty for His endless blessings and guidance throughout my journey. Without His grace, this work would not have been possible. I would like to express my deepest gratitude to my supervisor, **Prof. Ruchika Malhotra**, for her invaluable guidance, support, and encouragement throughout this research work. Her vast knowledge and expertise have been instrumental in shaping the direction and success of this research work. Throughout my journey, she provided insightful feedback and thoughtful advice, helping me overcome numerous challenges. I am deeply thankful for her mentorship, which has greatly contributed to my academic and personal growth. This research would not have been possible without her continuous support and belief in my potential.

I extend my heartfelt thanks to my husband, **Dr. Md. Tipu Khan**, for his patience, understanding, and continuous motivation. His belief in me kept me going, even during the toughest moments. A special note of appreciation goes to my father **Mr. Ubaidur Rahman Khan** and my mother **Mrs. Zareena Khan**, whose love, encouragement, and sacrifices have been the foundation of my success. I am forever grateful for their unconditional support. I would like to express my sincere gratitude and profound respect for my grandparents, whose unwavering love, wisdom, and encouragement have had an immeasurable impact on my life. Their guidance and support have been a constant source of inspiration throughout my academic journey. Their belief in the importance of education has been a driving force behind my accomplishments.

I also owe a debt of gratitude to my entire **family** for their constant encouragement and support. They have all played a vital role in helping me achieve this milestone. Last but not least, I would like to thank my dear **friends** for their companionship, positivity, and assistance throughout this journey. Their support has been invaluable. Thank you all for being a part of this journey.

Kishwar Khan

Abstract

Software defect prediction is a vital research area focused on improving the reliability and maintainability of software systems. As these systems become increasingly complex, the demand for accurate predictive models to identify defect-prone components grows more critical. Despite significant advancements in the field, challenges such as imbalanced datasets, feature selection, and the fine-tuning of machine learning algorithms for optimal performance persist. This research tackles these challenges by developing and validating enhanced Machine Learning (ML) techniques specifically designed for software quality prediction. The primary goal is to elevate the performance of prediction models by addressing essential issues like feature selection, hyperparameter tuning, and data imbalance, thereby enhancing the accuracy and robustness of these models. The research is validated through systematic reviews, empirical studies, and the creation of frameworks and tools applicable in real-world software development settings.

The thesis is systematically organized into several phases, each concentrating on different aspects of software defect prediction. The initial phase involves conducting a comprehensive systematic literature review to identify the most effective feature selection and machine learning algorithms currently employed in software defect prediction. This review establishes a foundation for understanding the current state of the field and highlights gaps that this thesis seeks to address. Key research questions examined include determining the most valuable feature selection and hyperparameter tuning technique for predicting defect-prone modules and assessing the effectiveness

of various machine learning algorithms. In the subsequent phases, the research focuses on developing and validating software defect prediction models using a range of feature selection and extraction techniques such as Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA), Kernel-based Principal Component Analysis (K-PCA) and Autoencoders with Support Vector Machine (SVM) as base machine learning classifier.

One of the significant contributions of this thesis is the development of a model for cross-project defect prediction using ten feature reduction techniques and Machine learning classifiers. The research explores the use of cross-project validation techniques, which are essential for ensuring that predictive models can generalize across different software projects. This is particularly important in scenarios where project-specific characteristics may vary, such as differences in coding practices or project domains. We analysed the impact of five filter based Feature Subset Selection techniques namely best first, exhaustive search, genetic search, greedy step wise search and random search along with five Feature Reduction techniques namely gain ratio, symmetrical uncertainty, oneR, information gain and reliefF and a no feature selection configuration by utilising the predictive ability of five frequently used classification approaches. Each method undergoes thorough testing and comparison across diverse datasets to confirm its validity and real-world applicability.

The need for evolutionary feature selection techniques arises from the complexity and high dimensionality of data in many machines learning tasks, including software defect prediction. Traditional feature selection methods may struggle to efficiently explore the vast search space of possible feature combinations, often leading to suboptimal performance. Evolutionary techniques, inspired by natural selection, offer a robust solution by iteratively optimizing feature subsets to enhance model accuracy and reduce overfitting. To address this, the thesis explores and implements a novel Software defect prediction model based on a variant of Grey Wolf Optimisation paired with Synthetic Minority Oversampling Technique. This model is particularly valuable when dealing with large datasets, complex feature interactions, and the need for balancing multiple objectives, such as maximizing predictive accuracy while minimizing computational cost.

This thesis also addresses the issue of imbalanced data, a prevalent challenge in software defect prediction. Imbalanced datasets often cause traditional machine learning models to produce biased predictions. To mitigate this, the study explores and implements techniques like the Synthetic Minority Over-sampling Technique (SMOTE). These methods are assessed based on their effectiveness in enhancing the prediction of defect-prone modules while reducing false positives. Additionally, hyperparameter tuning is a key focus of this research. Achieving optimal model performance often requires careful adjustment of parameters, and this study employs various tuning methods, including evolutionary and Bayesian optimization, to determine the best parameters for each predictive model. The research systematically evaluates the impact of hyperparameter tuning, demonstrating that well-tuned models significantly outperform those using default settings.

In conclusion, this thesis contributes substantially to the field of software defect prediction by tackling critical challenges in predictive modelling. By developing and validating advanced machine learning techniques, this work improves the accuracy, robustness, and practical relevance of predictive models in software development. The models and insights generated through this research hold the potential to make a significant impact in both academic and industrial contexts, offering researchers and practitioners new approaches for enhancing software quality. Moreover, by reducing software defects, this study contributes to the development of more reliable and secure software systems, ultimately benefiting society by fostering safer and more efficient technological environments.

Contents

List of Tables

List of Figures

List of Publications

Papers Accepted/Published in International Journals

1. Ruchika Malhotra and Kishwar Khan, “A novel software defect prediction model using two-phase grey wolf for feature selection.”, *Cluster Computing*, 2024.

2. Ruchika Malhotra and Kishwar Khan, “OpTunedSMOTE: A Novel Model for Automated Hyperparameter Tuning of SMOTE in Software Defect Prediction”, *Intelligent Data Analysis*, 2024.

Papers Accepted/Published in International Conferences

3. Ruchika Malhotra and Kishwar Khan, “A Study on Software Defect Prediction using Feature Extraction Techniques”, *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, Noida, India, 2020.

4. Ruchika Malhotra and Kishwar Khan, “Hyperparameter Optimization in Deep Learning for Improved Software Defect Prediction: A Stacked Ensemble Approach”, *5th Congress on Intelligent Systems (CIS 2024)*, Bengaluru, India, 2024.

Papers Communicated in International Journals

5. Ruchika Malhotra and Kishwar Khan, "A Systematic Review of Feature Reduction Techniques for Software Quality Predictive Modelling using Object-Oriented Metrics." *Arabian Journal for Science and Engineering.*

6. Ruchika Malhotra and Kishwar Khan, "Investigating the Effect of Feature Selection on Cross Project Defect Prediction", *International Journal of System Assurance Engineering and Management*.

Abbreviations

ML	**Machine Learning**
PCA	**Principal Component Analysis**
K-PCA	**Kernel-based Principal Component Analysis**
SMOTE	**Synthetic Minority Oversampling Technique**
KNN	**K-Nearest Neighbors**
RF	**Random Forest**
GB	**Gradient Boosting**
AdaBoost	**Adaptive Boosting**
XGB	**Extreme Gradient Boosting**
CNN	**Convolutional Neural Network**
LSTM	**Long Short-Term Memory**
GRU	**Gated Recurrent Unit**
SVM	**Support Vector Machine**
ROC-AUC	**Receiver Operating Characteristic - Area Under the Curve**
MCC	**Matthews Correlation Coefficient**
RBF	**Radial Basis Function**
CFS	**Correlation-based Feature Selection**

AMMIC	**Ancestor class MethodâMethod Import Coupling**
DMMEC	**Descendant class MethodâMethod Export Coupling**
OMMEC	**Other class MethodâMethod Export Coupling**
OMMIC	**Other class MethodâMethod Import Coupling**
FSS	**Feature Subset Selection**
FR	**Feature Ranking**
FS	**Feature Selection**
2M-GWO	**Two-phase Grey Wolf Optimizer**
GAN	**Generative Adversarial Network**
CTGAN	**Conditional GAN**
WGANGP	**Wasserstein GAN with Gradient Penalty**
SGD	**Stochastic Gradient Descent**
AUC	**Area Under the Curve**
DL	**Deep Learning**
BO	**Bayesian Optimization**
RQ	**Research Question**
HPT	**Hyperparameter Tuning**
LOC	**Lines of Code**
NOA	**Number of Attributes**
NOM	**Number of Methods**
NOC	**Number of Children**
DIT	**Depth of Inheritance Tree**
CBO	**Coupling Between Object Classes**
RFC	**Response for a Class**

LCOM	**Lack of Cohesion of Methods**
OO	**Object-Oriented**
SQA	**Software Quality Assurance**
SDP	**Software Defect Prediction**

Chapter 1

Introduction

1.1 Introduction

Software Quality Assurance (SQA) teams hold a vital position in the software development lifecycle, focusing on eliminating software defects. Consequently, many modern software organizations maintain a specialized SQA department [4]. Software engineers are responsible for carrying out numerous SQA tasks. Primarily, they create, implement, and perform tests to ensure that software systems fulfill their functional and non-functional requirements, align with user expectations, and achieve the necessary quality standards before being released to end-users[5, 6]. Additionally, they examine designs, assess code quality and associated risks, and refactor code to enhance its testability. Previous studies highlight concerns that SQA activities can be both costly and time-intensive. For instance, Alberts et al. highlight that SQA activities consume nearly 50% of the resources allocated for software development. Consequently, with constrained SQA resources, such as limited team size and time, it becomes impractical to thoroughly test and review an extensive software product[7].

A case study on the Mozilla project conducted by Mantyla et al. [8] demonstrates that adopting rapid release software development considerably escalates the workload of software testers. As a result, insufficient testing may lead to software defects, potentially incurring financial losses worth billions of dollars. In this context, defect prediction models are essential for effectively prioritizing SQA efforts. Software defect prediction models utilize historical data to detect software modules prone to defects. From an SQA standpoint, these models fulfill two primary objectives. Firstly, they can forecast which modules are likely to exhibit defects in the future[9–17].

SQA teams can leverage defect prediction models in a predictive context to allocate their limited resources efficiently, focusing on modules with the highest likelihood of being defective. Secondly, these models can help analyze the influence of different software metrics on a module's susceptibility to defects [13, 18–20]. The insights gained from defect prediction models enable software teams to avoid repeating mistakes associated with previously defective modules. Over the past decade, there has been a significant increase in the practical adoption of these models. Recent adopters of defect prediction models include, but are not limited to, the following examples: Bell Labs [13] , AT&T [21], Turkish Telecommunication [22], Microsoft Research [23–27] , Google [28], Blackberry [29], Cisco [30], IBM [31] , and Sony Mobile [32]. These organizations frequently share their success stories and insights gained from implementing defect prediction models.

1.1.1 Software Defect

A software defect, often called a bug, is defined as a flaw or deficiency in a software product that leads to unexpected behavior or performance[33]. The IEEE Standard 1044, Classification for Software Anomalies, offers a standardized vocabulary for terms relevant in this context. According to the standard:

- Defect: An imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced.

- Error: A human action that produces an incorrect result.
- Failure: Termination of the ability of a product to perform a required function or its inability to perform within previously specified limits. Or an event in which a system or system component does not perform a required function within specified limits.
- Fault: A manifestation of an error in software.

Since software defects can cause the entire embedded system to malfunction, potentially endangering health or even lives in safety-critical systems, most software development organizations strive to release software without known defects. All defects identified during verification and validation processes are meticulously recorded in a defect database. These databases, which can be specific to a team, project, product, or organizational division, follow a predefined format to ensure consistency. Their primary purpose is to facilitate the resolution of reported defects. The database typically serves as a platform where various stakeholders, both within and outside the organization, can:

- Access the information about defect(s) of their interest,
- Add, edit, or update the information related to a given defect,
- Comment, provide expertise or guidance to help resolve the defect, and
- Track the progress of reported defect(s) and monitor defect statistics.

To streamline the documentation and sharing of information, multiple attributes are recorded for each reported defect. Some attributes are mandatory, offering essential details about the defect, while others are optional, providing supplementary information. The primary objective is to facilitate communication between the actor (typically the tester) who identified the defect and the actor(s) (usually developers) responsible for resolving or assisting in its resolution.

1.1.2 Software Defect Prediction

A software defect is a noticeable anomaly that may cause a software system to fail in performing its intended function. Often referred to as a defect, it is typically identified by a tester and is commonly labeled as a bug, defect, or error [34]. According to the definition provided by ISO/IEC/IEEE 24765, a software defect can be described as an apparent indication of an error within a software system. Software defect prediction (SDP) is a systematic procedure that effectively identifies the software module or class that has a higher probability of containing defects [35, 36].

Developers are required to manage continuous changes, adhere to strict time constraints, and ensure flawless functionality in their deliverables [2]. Nevertheless, the constraints of limited human resources and time pose significant challenges to the testing of software systems. Therefore, it is imperative to allocate available resources in an efficient manner, giving priority to the source code that potentially contains more defective modules, thereby requiring a greater allocation of testing resources. As an illustration, consider the cost of a Java-based project [37] with a size of 100 function points and 10 KLOC. The allocation of effort for the five distinct phases is as follows: 10% is allocated for requirement engineering, analysis, and design; 20% is allocated for coding; 30% is allocated for testing; and 5% is allocated for installation and training. Accurate prediction involves focusing attention towards modules that are prone to defects that require more extensive testing. This approach helps in effectively prioritizing and allocating limited testing resources, thereby reducing the overall testing effort.

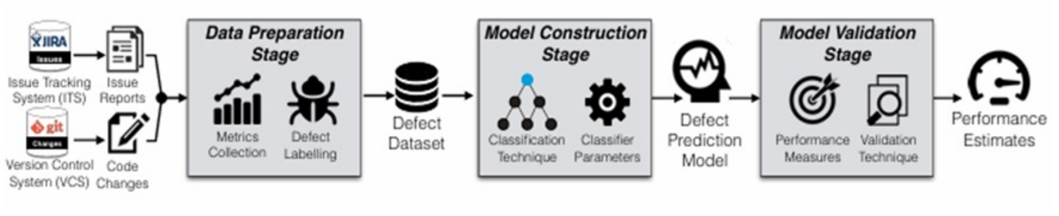


Figure 1.1: An overview of the typical defect prediction modelling and its related experimental components.

The extensive adoption of software systems across various industries underscores the potentially severe consequences of software defects, which can pose significant risks to human well-being and incur substantial financial burdens. For instance, in the UK, as many as 200 fatalities each year are attributed to avoidable defects in patient care systems [38]. The SDP method generally involves a supervised approach, wherein a set of independent variables (also known as predictors) are utilized to make predictions about the dependent variable (specifically, the defect-prone module). A model is trained using a combination of machine learning techniques [39], deep learning techniques [40], and statistical learning techniques [41].

Figure 1.1 depicts the fundamental architecture of the SDP technique. To extract software defects, the initial step involves organizing a metrics dataset comprising software modules, encompassing attributes like module size and module complexity. These metrics are typically obtained from a version control system. Furthermore, it is essential to assign appropriate labels to defective modules in the event that they have been influenced by changes in the code aimed at resolving a reported issue that falls under the category of a defect. Furthermore, defect models are trained using a machine learning technique. Moreover, it is necessary to appropriately configure the parameter settings of machine learning techniques in order to regulate their characteristics. After completion of training, the model that completed training is capable of making predictions regarding the classification of various modules (whether they are faulty or non-faulty). These predictions are applicable to both new projects and target projects. The training of a prediction model can be classified into two distinct types, i.e., Homogeneous software defect prediction and Heterogeneous defect prediction. Homogeneous software defect prediction is further categorized into two distinct types. First, the Within-Project Defect Prediction (WPDP) model in which a predictive model is developed by utilizing labelled instances from Project A. This model is then employed to classify unlabelled instances within the same project as either defective or clean. Second, The Cross-Project Defect Prediction (CPDP) involves the training of a prediction model using labelled instances from Project A (referred to as the source) in order to predict defects in Project B (referred to as the target). In CPDP, the source and target projects generated by a prediction combination indicate similar metric sets. The Heterogeneous defect prediction (HDP) is similar to CPDP, except the prediction combination involves source and target projects that possess distinct sets of metrics. Next, performance measures are chosen to evaluate the effectiveness of defect prediction models. At last, the models are validated to assess their performance when applied to novel software modules.

1.2 What is Predictive Modelling?

Predictive modelling is a statistical technique used to create a model that can forecast future outcomes based on historical data. It involves identifying relationships between variables in a dataset to predict unknown or future values of a target variable. This approach is widely used in various fields such as finance, marketing, healthcare, and software engineering. The core idea is to learn from past occurrences by training a model that can generalize and make accurate predictions on new data. Common techniques used in predictive modelling include regression analysis, decision trees, neural networks, and machine learning algorithms.

In software engineering, predictive modelling plays a crucial role in quality assurance processes, such as software defect prediction. By analyzing historical data related to software modules, such as their size, complexity, and previous defect records, predictive models can estimate which components are likely to be prone to defects. This helps allocate testing resources more efficiently and prioritize modules that require more attention, ultimately reducing the cost and effort involved in software maintenance.

A key challenge in predictive modelling is ensuring the accuracy and generalizability of the model. Techniques like cross-validation, feature selection, and hyperparameter tuning are often applied to optimize the model's performance. Additionally, managing issues such as imbalanced datasets and overfitting is critical to developing reliable models. When applied effectively, predictive modelling can provide valuable insights and enable organizations to make data-driven decisions, leading to better outcomes and reduced risks.

1.2.1 Steps in Predictive Modelling

The following are the steps that are used in Predicting modelling and are shown in Figure 1.2 .

1. **Objective of problem under analysis:** The first step is to define the objectives of the application that needs to be analysed. For instance, a company wants to predict the sales of a particular product so every parameter is stated in this step.
2. **Defining analytical goals:** The goals are finalised after completing the feasibility study of the problem domain.
3. **Data preparation:** In this step, analysis team understand the data and prepare data for analysis. The predictive model would be efficient if the underlying

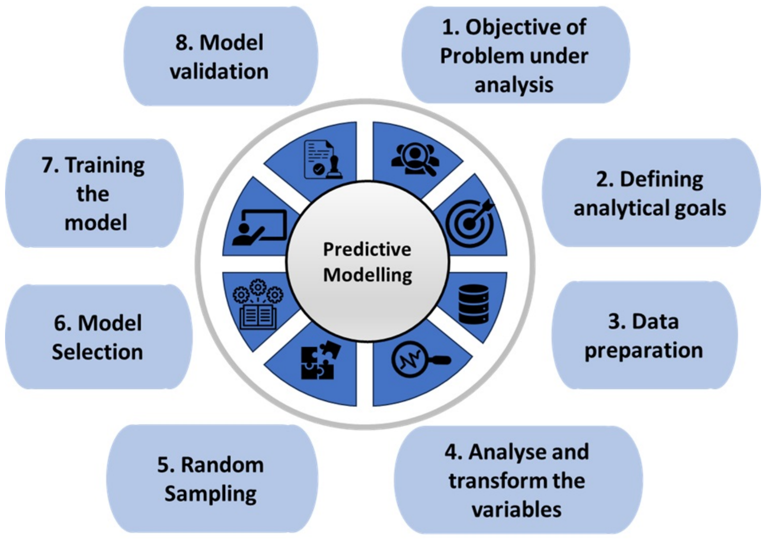


Figure 1.2: Steps in Predictive Modelling

data is good enough for the analysis. The cleaning of data is done in this step where data is converted from imbalance to balance data. The data is formatted according to the tool selected by the analysis team and null/missing values are dealt properly.

4. **Analyse and transform the variables:** Here, the null and missing values are dealt properly and dimension are reduced using various techniques like principle component analysis,factor analysis, etc.

5. **Random sampling:** The data is divided into training and testing set depending on the ratio selected.

6. **Model selection:** Depending on the goals of the application, models are selected either supervised or unsupervised.

7. **Training the model:** Models are trained by feeding them with the data prepared at step 3.

8. **Validation:** In the final step, results are validated using cross validation so that accuracy can be improved.

   As more and more data are being produced, it is important to generate a model that will be able to understand that data thoroughly. Predictive modelling steps forward and helps the organisation in making decision based on data.

1.2.2 Predictive Modelling for Software Defect Prediction

Predictive modelling plays a crucial role in software defect prediction, helping software engineers and managers anticipate the defect-prone areas of a software system before they occur in production. In the broader context, predictive models are used across various domains to forecast outcomes based on historical data, and in the software engineering fi eld, they are used to predict software quality attributes like defect-proneness. Over the years, numerous models have been developed to support the prediction of defects, enabling software teams to take preventive measures before releasing software. These models are essential for early detection of defects, which can drastically reduce the cost of fixing bugs compared to addressing them later in the software development lifecycle.

Predictive models for software defect prediction are built using historical defect data and software metrics such as complexity, size, coupling, cohesion, and other internal attributes of the code. By understanding the correlation between these metrics and the likelihood of defects, the models can predict which parts of the software are more likely to contain bugs. This helps managers and developers focus their testing and quality assurance efforts on high-risk areas, leading to more efficient use of limited resources like time, budget, and human effort. Implementing predictive models early in the software development process allows for better planning, prioritizing, and resource allocation for testing. It also aids in risk management by focusing attention on areas of the software that are most likely to fail. Techniques for developing these models range from statistical methods to more advanced approaches such as machine learning, search-based techniques, and hybrid methods. These models continuously learn from new data, refining their predictions for future versions of the software product.

By incorporating predictive modelling into defect prediction, software teams can enhance the overall quality of their products, reduce time-to-market, and improve customer satisfaction by delivering more reliable software.

1.2.3 Factors Affecting the Performance of Predictive Modelling for Software Quality

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

- **Feature Selection**: Feature selection is the process of identifying the most relevant features that contribute to the predictive power of the model. In the context of software quality prediction, 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 depends on the nature of the data, the predictive task, and the specific requirements of the model.

- **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.

- **Imbalanced Data**: Imbalanced data is a common challenge in predictive modelling for software quality, particularly in tasks such as defect prediction and defect categorization. Imbalanced data occurs when the distribution of the target variable 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.

1.3 Literature Survey

Software defect prediction plays a critical role in ensuring the delivery of high-quality software, as it allows for early identification of potential defects that can compromise functionality and user satisfaction. Over the years, researchers have focused extensively on enhancing techniques and methodologies to improve software defect prediction accuracy and efficiency. Conducting a thorough review of existing literature in this domain is vital to identifying gaps in current research, discovering areas for innovation, and driving further advancements in the field.

This section explores various software metrics commonly utilized in the literature for predicting software defects. It also reviews a wide range of models proposed by researchers to predict software defects, with a focus on the methodologies employed. These models have been developed using diverse machine learning techniques, each contributing unique insights into improving the precision of software defect predictions.

1.3.1 Software Metrics

Software metrics can be considered as a quantitative measurement that assigns symbols or numbers to features of predicted instances [42]. In fact, they are features, or attributes, that describe many properties such as reliability, effort, complexity, and quality of software products. These metrics play a key role in building an effective software defect predictor. They can be divided into two main categories: code metrics and process metrics [43].

1.3.1.1 Code Metrics

Code metrics, also called product metrics, are directly collected from existing source code. These metrics measure the complexity of source code based on the assumption that complex software components are more likely to contain bugs. Throughout the history of software engineering, various code metrics have been used for software defect prediction.

Size: The first metric is the size metric introduced by Akiyama (1971)[9]. In order to predict the number of bugs, the author uses the number of lines of code as the only metric. Numerous software defect prediction studies have applied this metric for building predictors [20, 34, 36, 44, 45]However, using only this metric is too simple to measure the complexity of software products.

Halstead and McCabe: For this reason, other useful, widely used, and easy-to-use metrics have been applied for creating defect predictors [34, 36, 46]. These metrics are called static code attributes introduced by McCabe (1976) and Halstead (1977) [47]. Halstead attributes are selected based on the reading complexity of source code. They are defined using several basic metrics collected from a software instance, including:

Table 1.1: Halstead basic measurement [1].

Symbol	Description
μ_1	Number of distinct operators
μ_2	Number of distinct operands
N_1	Total number of operators
N_2	Total number of operands
μ_1^*	Minimum possible number of operators
μ_2^*	Minimum possible number of operands

The first four metrics are self-explanatory, whereas μ_1^* and μ_2^* are potential operator and operand counts in a software instance. For example, $\mu_1^* = 2$ is the minimum number of operators for a default function with the function's name and a grouping symbol, while μ_2^* is the number of parameters passed to the function, with no repetition.

The Halstead metrics defined using the above metrics include:

Table 1.2: Halstead Metrics

Name	**Description**
Length: $N = N_1 + N_2$	The program length
Vocabulary: $\mu = \mu_1 + \mu_2$	The vocabulary size
Volume: $V = N \times \log_2 \mu$	The information content of a program
Potential volume: $V^* = (2 + \mu_2^*) \times \log_2(2 + \mu_2^*)$	The volume of the minimal size implementation of a program
Level: $L = V^*/V$	The program level
Difficulty: $D = 1/L$	The difficulty level of a program
Error estimate: $\hat{I} = \frac{2}{\mu_1} \times \frac{\mu_2}{N_2}$	Error estimate for a program
Content: $I = \hat{I} \times V$	The intelligence content of a program
Effort: $E = \frac{V}{L} = \frac{\mu_1 N_2 N \log_2 \mu}{2\mu_2}$	The effort required to generate a program
Programming time: $T = \frac{E}{18}$ (seconds)	The programming time required for a program

1.3.1.2 McCabe Attributes

McCabe attributes are cyclomatic metrics representing the complexity of a software product. These attributes are based on the assumption that âthe complexity of pathways between module symbols is more insightful than just a count of the symbolsâ [36]. Differing from Halstead attributes, McCabe attributes measure the complexity of source code structure. They are obtained by computing the number of connected components, arcs, and nodes in control flow charts of source code.Each node of the flow chart represents a program statement while an arc is the flow of control from a statement to another. The following are three complexity attributes introduced by McCabe (1976) [48].

- **Cyclomatic complexity**: Denoted by $\nu(G)$, represents the number of linearly independent paths through the flow chart. $\nu(G) = e - n + 2$ in which G is the flow chart, e represents the number of arcs, and n is the number of nodes [1].

- **Essential complexity**: Denoted by $e\nu(G)$, measures the degree to which a flow chart is able to reduce by decomposing all the sub flow charts that are proper one-entry, one-exit, or D-structured primes [42]. $e\nu(G) = \nu(G) - m$, in which m is the number of sub flow charts of G.

- **Design complexity**: Denoted by $i\nu(G)$, represents the cyclomatic complexity of a reduced flow chart of a class or module. The reduction is done to remove complexities that do not affect the interrelationship between design classes or modules [47].

1.3.1.3 Object-Oriented Metrics

In fact, such inter-class metrics have been produced by Henry and Kafura (1981)[49] with fan-in metrics that represent the number of software components invoking a given component, and fan-out metrics that represent the number of software components invoked by a given component. Besides fan-in and fan-out, other metrics measuring quantity and volume of source code have also been introduced (D'Ambros, Lanza & Robbes, 2012)[44].

As listed in Table 3, several of these metrics are quite simple to compute by counting the number of public and private attributes and methods. In practice, these metrics are designed based on characteristics of object-oriented models including inheritance, reusability, cohesion, encapsulation and coupling. Therefore, the collection of these metrics, also known as object-oriented (OO) metrics, is suitable for evaluating object-oriented systems. With the popularity of object-oriented programming, OO metrics are becoming increasingly widely used for building software defect prediction models. Many of such models have been proposed by DâAmbros et al. (2012); Kim, Zhang, Wu and Gong (2011); Lee, Nam, Han, Kim and Hoh (2011); Pai and Dugan (2007); Wu, Zhang, Kim and Cheung (2011); Zimmermann and Nagappan (2008);

Pan and Yang (2010)[17, 44, 45, 50–52].

Table 1.3: List of class-level object-oriented metrics [2].

Name	Description
Fan-in	The number of other classes that reference the measured class
Fan-out	The number of classes referenced by the measured class
NOA	The number of attributes
NOPA	The number of public attributes
NOPRA	The number of private attributes
NOAI	The number of attributes inherited
LOC	The number of lines of code in a class
NOM	The number of methods
NOPM	The number of public methods
NOPRM	The number of private methods
NOMI	The number of methods inherited

Apart from these OO metrics, several other metrics have been empirically proven to be effective for predicting defects in object-oriented programs. In 1994, Chidamber and Kemerer introduced a set of CK metrics, which have been widely used in numerous studies to create software defect predictors [44, 50, 51, 53, 54].

Table 1.4: CK metrics [3]

Name	Description
WMC	Weighted methods per class
DIT	Depth of inheritance tree
NOC	Number of children
CBO	Coupling between object classes
RFC	Response for a class
LCOM	Lack of cohesion of methods

The metrics listed in Table 4 can be described as follows:

- **Weighted methods per class (WMC)**: This metric measures the complexity of an individual class. It is a weighted sum of all methods in a class.

- **Depth of inheritance tree (DIT)**: This metric measures the length of the longest path of inheritance ending at a class. If the inheritance tree for the measured class is deeper, it is more difficult to estimate the behaviour of the class.
- **Number of children (NOC)**: This metric counts the number of immediate child classes that inherit from the current class.
- **Coupling between object classes (CBO)**: This metric measures the dependency of a class on others by counting the number of other classes coupled to the measured class. A class is coupled to others if it invokes variables or functions of the other classes.
- **Response for a class (RFC)**: This metric counts the number of methods potentially executed in response to a message received by an object of a class.
- **Lack of cohesion of methods (LCOM)**: This metric is the subtraction of the number of method pairs sharing no member variable from the number of method pairs sharing at least one member variable.

1.3.2 Software Defect Prediction

Software defect prediction is a critical process in software engineering that aims to identify modules likely to contain defects before they are released. This proactive approach allows project managers to allocate testing resources more effectively, ultimately enhancing software quality and reducing costs. SDP utilizes various metrics derived from software code, such as method-level and class-level metrics, to assess the likelihood of defects in software modules or not [55].

The process of SDP faces several significant challenges that hinder the development of effective predictive models. Key issues include data quality and availability, as historical data often suffers from incompleteness, noise, and bias, leading to unreliable predictions [56, 57]. Class imbalance, where non-defective instances vastly outnumber defective ones, results in biased models with poor defect detection rates [58]. Identifying relevant features from high-dimensional datasets is crucial yet difficult, as improper selection can lead to overfitting. Additionally, complex machine learning models often lack interpretability, making it hard for stakeholders to trust and utilize predictions [59]. Generalizing models across different projects is challenging due to variations in codebases and practices, and evolving software environments require continuous model updating to remain effective. Furthermore, integrating defect prediction models into existing development processes poses logistical and technical hurdles. HPT is another critical issue, as selecting optimal parameters is essential for maximizing model performance but can be computationally intensive and complex. Addressing these challenges through improved data quality, class imbalance techniques, enhanced model interpretability, better generalization methods, and effective HPT is essential for advancing SDP's applicability and effectiveness in real-world scenarios.

1.3.3 Class Imbalance Problem

The class imbalance problem is a significant challenge in SDP models. It occurs when there is a disproportionately large number of non-defective instances compared to defective instances in the training data. This imbalance can lead to poor performance of traditional machine learning algorithms, as they tend to get biased towards the majority non-defective class.

Researchers have attempted to address this issue using sampling methods, ensemble methods, and cost-sensitive methods. Sampling methods include oversampling techniques like SMOTE and ROS and undersampling techniques like RUS. Studies have shown that while these methods improve the performance of linear and logistic models, neural networks and classification tree models may underperform. Ensemble methods, such as bagging, boosting, and stacking, create and combine several weak learners to form a strong classifier, with variations like SMOTEBagging and RUSBoost demonstrating effectiveness in SDP [60]. Cost-sensitive learning, which uses a cost matrix to define different misclassification costs, has also been applied with methods such as cost-sensitive KNN and neural networks, showing improved prediction performance. Generative Adversarial Networks (GAN) methods, particularly Vanilla GAN, Conditional GAN (CTGAN), and Wasserstein GAN with Gradient Penalty (WGANGP), have shown superior performance in SDP on imbalanced datasets compared to traditional techniques like SMOTE and ROS. These methods generate synthetic instances that closely resemble real instances, improving data quality and diversity. The study highlights the effectiveness of using generative models such as GANs, variational autoencoders, and adversarial autoencoders for cross-project defect prediction (CPDP), with the GAN model combined with the Stochastic Gradient Descent (SGD) classifier yielding the best results. Modifications to the GAN architecture to suit numerical data further enhanced performance, resulting in marginal improvements in accuracy, precision, recall, and F1-score. Despite limited improvements from hyperparameter optimization and undersampling combinations, the GAN-based methods consistently outperformed other oversampling techniques in various metrics, including the Area Under the Curve (AUC) for Decision Tree and Random Forest classifiers across multiple datasets[61–63].

One of the most popular techniques to address the class imbalance problem is SMOTE. It generates new synthetic instances of the minority (defective) class by interpolating between existing minority class instances that lie together. This helps to increase the representation of the minority class and reduce the imbalance ratio[64, 65]. In this study, we focused on the SMOTE technique to balance the dataset.

1.3.4 Hyperparameter Tuning

Hyperparameter tuning plays a crucial role in enhancing the performance of SDP models. Properly configured hyperparameters can significantly improve the accuracy and efficiency of these models, which are essential for identifying potential defects in software systems.

Hyperparameters must be set prior to training and can include parameters such as the number of trees in a random forest, the learning rate in gradient boosting, or the number of neighbours in k-nearest neighbours. The choice of hyperparameters can dramatically affect the model's performance, making tuning essential for optimal results. Research indicates that HPT can lead to significant improvements in the performance of various machine learning algorithms used in SDP[66].

Nowadays, researchers are focusing on the HPT of preprocessing techniques as well[67]. HPT of preprocessing techniques involves selecting the optimal configuration for the parameters that control the preprocessing steps applied to data before it is fed into a machine learning model. This process is crucial for improving model performance, as the choice and configuration of preprocessing steps can significantly impact the quality and predictive power of the data. Tuning ensures that the data is represented in the most informative way, which can enhance the learning process of the model.

1.3.5 Feature Selection

Catal and Diri [67] investigated the influence of dataset size, metrics collectors, and FS approaches on SDP. To make the predictive models repeatable, refutable, and verifiable, they employed publicly available NASA data from the open-source PROMISE repository. According to this analysis, random forests have the best prediction results for large datasets. For datasets of limited size, Naive Bayes is the most predictive approach in terms of the ROC-AUC assessment metric. When method-level metrics are being used, the parallel version of the Artificial Immune Recognition Systems method seems to be the strongest paradigm-based algorithm. By integrating feature selection and the approach of data sampling, Gao and Khoshgoftaar [68] suggested a methodology for dealing with high dimensionality and class imbalance. The data sets used in this study were retrieved from the PROMISE repository. The findings indicate that the model built with sampled data and a reduced feature set performed better than the model built on FS on original data. Wang et al. [69] provided an empirical analysis of six frequently used filter-based software metric rankers, as well as our suggested ensemble methodology based on attribute rank ordering (mean or median), which was applied to three major software projects using five widely used learners. The ROC-AUC evaluation metric was used to assess classification accuracy. The results show that the ensemble approach outperformed any single ranker in terms of average performance and robustness. Variations among rankers, learners, and software projects had a major impact on classification outcomes, according to the empirical analysis, and the ensemble method will smooth out results.

In order to predict software defects, Tumar et al. suggested an enhanced and binary version of Moth Flame Optimisation (MFO). The suggested approach improves the results from the literature and demonstrates the significance of transfer functions for FS methods [70]. Turabieh et al. [71]in their study discusses software Defect prediction (SDP) and proposes a novel approach to amplify the effectiveness of a layered recurrent neural network (L-RNN) employed as a binary classifier in SDP. The approach involves using binary versions of Genetic Algorithm based optimiser, Particle Swarm based Optimisation and Ant colony-based optimisation for FS algorithms to remove unnecessary attributes and enhance the effectiveness of the L-RNN. This paper compares the suggested methodology with other modern methods using 19 real-life software projects from the open-source repository. The findings demonstrate that the suggested strategy works comparatively better than other current approaches.

1.4 Objectives of the Thesis

1.4.1 Vision

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

1.4.2 Focus

The focus of the thesis is centred on several key research objectives, each addressing a specific aspect of software defect prediction modelling. These objectives are designed to systematically explore, analyse, and improve the factors that influence the performance of software defect prediction models, with a particular emphasis on feature selection and parameter tuning. The research is also focused on developing novel and improved classification models that are effective in identifying defect-proneness, thereby enhancing the predictive accuracy and reliability of software defect prediction models. 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. Thus, this study explicitly addresses the following objectives:

The proposed work has the following main objectives:

- To perform a systematic literature review in order to gain insights into the effect of feature selection techniques on software quality prediction models.
- To investigate the effect of feature selection on cross-project defect prediction.

- To explore evolutionary feature selection algorithms and evaluate their effectiveness for developing models for software defect prediction.
- To develop novel software defect prediction models using feature selection and machine learning techniques.
- To analyse the effect of parameter tuning techniques for software defect prediction models.

1.4.3 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 feature selection techniques employed for software quality prediction models**
   - Study of existing research publications that would help to understand the process and procedure of developing feature selection-based models for predicting software defects.
   - An extensive study of existing literature to understand the significance of feature selection for software defect prediction.
   - Study of various machine learning and statistical methods used in the literature for developing software defect prediction models with their strengths and weaknesses.
   - A review of the literature would also help to gain insights into both the quantitative and qualitative perspectives of predictive modelling and help

to identify the problem areas and gaps in the existing literature.

2. **To investigate the effect of feature selection on the cross-project defect prediction**

    - To assess the significance of feature selection-based models on cross-project defect prediction.
    - To evaluate the competency of an ensemble of classifiers for cross-project defect prediction and compare them with prevailing machine learning and statistical techniques.
    - To compare and statistically validate the results using some statistical tools.

3. **To explore evolutionary feature selection techniques and evaluate their effectiveness for developing models for software defect prediction**

    - To assess the applicability of evolutionary-based feature selection algorithms for software defect prediction models.
    - To compare the performance of proposed evolutionary-based feature selection techniques with other feature selection techniques to discover the most suited algorithm for developing software defect prediction models.
    - To analyse the evolutionary feature selection techniques using various validation techniques in order to produce unbiased and generalised results.

4. **To develop a novel software defect prediction model using feature selection and machine learning techniques**

    - To propose a novel feature selection-based model for efficiently predicting defects in software.

- To utilise different machine learning techniques along with the feature selection and compare their performances.
- To explore which metrics (features) are mostly selected by feature selection techniques and how these metrics are related to software defects.

5. **To analyse the effect of parameter tuning techniques on software defect prediction models**

    - To evaluate and compare different Machine Learning and Deep Learning-based software defect prediction models by optimizing their parameters.
    - To investigate the impact of fine-tuning parameters in both preprocessing stages and classifiers on the performance of software defect prediction models.
    - To assess the sensitivity of various classifiers to parameter tuning and its influence on defect prediction accuracy.
    - To examine the trade-off between parameter tuning and computational efficiency, identifying techniques that offer the best balance between performance and resource utilization.

1.5 Organization of the Thesis

This section presents the organization of the thesis. The thesis is structured into eight chapters, each focusing on a specific aspect of the research undertaken to develop and validate improved machine learning techniques for software defect prediction. The organization of the thesis is as follows: **Chapter 1** sets the stage for the entire research work by presenting the basic concepts of the work and the motivation behind 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 research gaps. **Chapter 4** discusses the construction of software defect prediction models focusing on feature selection and extraction. **Chapter 5** examines the impact of feature selection techniques on software defect prediction. **Chapter 6** proposes a novel software defect prediction model based on an evolutionary algorithm for feature selection. **Chapter 7** introduces two software defect prediction models that integrate hyperparameter tuning with feature selection. Finally, **Chapter 8** presents the conclusions of the thesis. A brief description of each chapter is provided below.

Chapter 1: This chapter sets the stage for the entire research work by discussing the significance of software defect prediction in the context of software engineering. It outlines the challenges associated with predictive modelling in software quality, such as the need for efficient feature selection, parameter tuning, and handling imbalanced data. The chapter also presents the research objectives, the scope of the study, and the contributions of the research. Additionally, it describes the detailed steps involved in developing software defect prediction models.

Chapter 2: This chapter details the research methodology adopted to achieve the research objectives. It begins with a discussion on the study's design, 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 pre-processing techniques, and the steps involved in developing and validating predictive models. Finally, it 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 the effects of feature selection in 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 constructing software defect prediction models, specifically examining feature selection and extraction techniques. It provides an in-depth comparison of these techniques, assessing their effectiveness across multiple datasets. The chapter offers valuable insights into selecting suitable feature extraction methods and lays the groundwork for future exploration in the field.

Chapter 5: This chapter delves into the impact of feature selection techniques on software defect prediction models, particularly in scenarios with limited historical data. It explores cross-project defect prediction (CPDP) techniques, employing multiple machine learning classifiers and feature selection algorithms. The findings highlight the importance of combining feature selection with advanced algorithms to enhance prediction accuracy, with statistical analyses confirming the significance of the results.

Chapter 6: This chapter introduces a novel model for software defect prediction, focusing on challenges like redundant and irrelevant features in datasets. The proposed model utilizes a variant of the Grey Wolf Optimizer for feature selection, combined with the Synthetic Minority Oversampling Technique (SMOTE) to balance the dataset. The chapter presents experimental results demonstrating the model's superior performance in improving prediction accuracy, validated by statistical techniques.

Chapter 7: This chapter presents two significant studies advancing software defect prediction. The first study proposes an optimized approach to handling imbalanced data and hyperparameter tuning using the Tree-structured Parzen Estimator algorithm within the Optuna framework. The second study develops a stacked-ensemble model utilizing deep learning techniques, such as convolutional neural networks and long short-term memory networks. Both studies underscore the critical role of optimization in enhancing the reliability and effectiveness of software defect prediction models.

Chapter 8: 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

Achieving the objectives of this research and ensuring reliable empirical results requires a carefully structured approach. The research methodology serves as a blueprint that provides a systematic sequence of steps, guiding the design, execution, and analysis phases of empirical experiments. By adhering to a structured methodology, the research gains consistency, reduces bias, and enhances the reproducibility and validity of findings. This chapter presents the research methodology employed in this thesis, offering a clear framework to address the research questions and validate the proposed methods. It delineates each stage of the process, from defining the research problem and reviewing relevant literature to selecting variables, analysing data, and developing the experimental design. Each step is designed to build upon the previous one, ensuring that the overall research process is cohesive, logical, and aligned with the research objectives.

The chapter is structured as follows: Section 2.2 outlines the research process employed for empirical experiments, followed by Section 2.3, which defines the research problem. Section 2.4 provides a comprehensive literature review, summarizing relevant studies. Section 2.5 specifies the independent and dependent variables used in this research, while Section 2.6 describes the data analysis techniques applied. Section 2.7 explains the data collection process, and Section 2.8 elaborates on the data preprocessing techniques. Section 2.9 focuses on feature reduction methods, and Section 2.10 addresses data balancing strategies. Section 2.11 discusses the development and validation of the prediction model, Section 2.12 reviews performance measures, and Section 2.13 concludes with statistical analysis techniques.

2.2 Research Process

The research process is a systematically planned sequence of steps designed to thoroughly explore and address the research problem. This structured approach ensures a logical flow, guiding each stage of the investigation from initial problem identification through to experimental evaluation and conclusion. Figure 2.1 visually represents the research process, illustrating the steps integrated throughout the chapters of this thesis to maintain consistency and clarity. Each stage within this process is crucial, contributing to a comprehensive and methodical investigation. The following sections delve into each step, providing a detailed explanation of the activities undertaken at each phase, how they relate to one another, and how they support the overall research objectives. Through this structured process, the research not only achieves coherence and direction but also fosters rigor and validity, ultimately enhancing the reliability of the empirical findings.

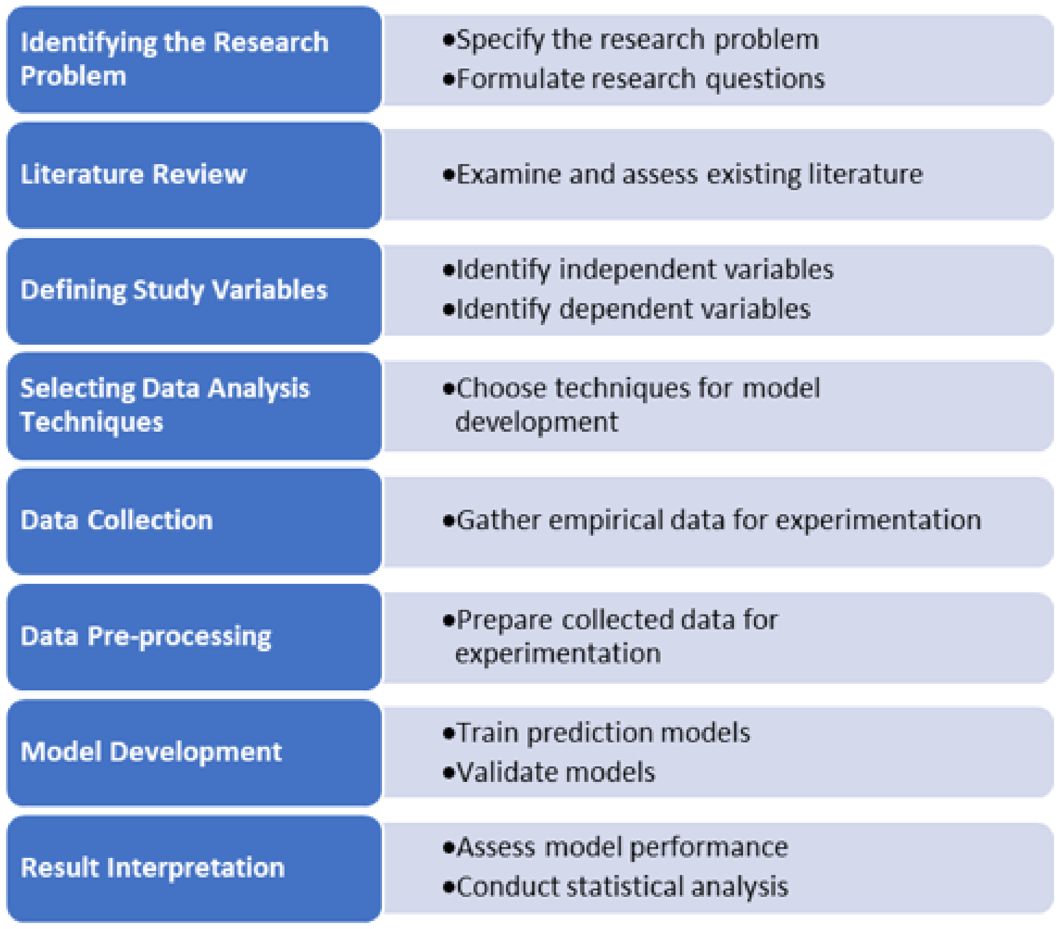

Identifying the Research Problem
•Specify the research problem
•Formulate research questions
Literature Review
•Examine and assess existing literature
Defining Study Variables
•Identify independent variables
•Identify dependent variables
Selecting Data Analysis Techniques
•Choose techniques for model development
Data Collection
•Gather empirical data for experimentation
Data Pre-processing
•Prepare collected data for experimentation
Model Development
•Train prediction models
•Validate models
Result Interpretation
•Assess model performance
•Conduct statistical analysis

Figure 2.1: Research Process

2.3 Identify the Research Problem

The initial step in the research process involves formulating the research problem. During this phase, the core issue under investigation is clearly articulated and defined in the form of specific research questions (RQs). These RQs serve as a guide, providing focus and direction to the research efforts. By defining precise RQs, the study establishes a framework for conducting experiments that aim to systematically explore and answer these questions.

In this thesis, the following research questions are addressed, each designed to contribute to a deeper understanding of the research problem and to achieve the study's objectives. These RQs shape the methodology, guiding the selection of data, analysis techniques, and experimental design to produce meaningful and reliable insights:

1. What is the current state of literature in the domain of Software Defect Prediction (SDP), and what research gaps exist related to feature selection?
2. How does the performance of SDP models based on feature selection vary when developed from imbalanced datasets?
3. What evolutionary-based feature selection techniques can researchers utilize to develop efficient SDP models?
4. How does parameter tuning perform in developing SDP models with feature selection?

2.4 Literature Review

To understand the research problem, a comprehensive literature survey of existing studies on software defect prediction (SDP) is essential. Through this review, we gain insight into the extent to which the SDP problem has been explored in the literature. Over the years, researchers have developed various models aimed at predicting software defects []. These models establish relationships between internal characteristics of software and defect proneness. Software metrics are used to capture these internal characteristics, and models have been developed using both commercial and open-source project datasets. Such models assist software practitioners in predicting defect-prone components early in the development stages. Identifying software defects early allows project managers to allocate resources efficiently, focusing on components with high defect potential. Consequently, the literature emphasizes that developing effective SDP models is crucial for enhancing software quality.

2.5 Defining Study Variables

In empirical investigations, there are two types of variables involved: independent (predictor) variables and dependent (response) variables. The dependent variable is the primary focus of the research. In this study, the dependent variable is software defects. This variable is influenced by the independent or predictor variables. Through these independent variables, the dependent variable can be predicted. Independent variables must remain unaffected by other variables in the analysis[55]. This thesis aims to develop SDP models that use independent variables to predict software defects, the dependent variable. Here, the independent variables are software metrics that represent various characteristics of software classes. We examine independent variables from widely recognized software metric suites to explore their relationship with software defects.

2.5.1 Independent Variables

The independent variables employed in software defect prediction models primarily consist of a wide range of software metrics. These metrics serve as the foundational features for analyzing and predicting the presence of defects within software systems. A comprehensive explanation of these metrics, including their definitions, classifications, and roles in defect prediction, is provided in section 1.3.1 of this thesis. This section delves into the specifics of each metric, offering detailed insights into their derivation and measurement.

2.5.2 Dependent Variables

In software defect prediction, the dependent variable also referred to as the target variable or response variableâis a crucial component of the modelling process, as it determines the outcome that the model aims to predict. This variable typically indicates the presence or absence of defects within a software module, file, or code unit, serving as the basis for evaluating the effectiveness of the predictive model [72].

One common approach is to use a binary classification for the defect status. In this case, the dependent variable categorizes components into two groups: defective (represented as 1) and non-defective (represented as 0). This binary framework is prevalent in traditional defect prediction studies, where the objective is to determine whether a particular component is likely to contain defects based on historical data and various software metrics. The simplicity of this approach makes it easy to interpret and implement, especially in contexts where the goal is to identify high-risk areas within the code base.

2.6 Data Analysis Methods

In this thesis, we have employed a variety of machine learning (ML), ensemble, and deep learning (DL) techniques to construct prediction models with enhanced accuracy and reliability. ML techniques encompass traditional algorithms that capture and model the underlying relationships between predictor variables and the target variable, often relying on mathematical equations. These algorithms, such as support vector machines (SVM) and k-nearest neighbours (KNN), learn from historical data by identifying patterns within the dataset, allowing them to represent and generalize the relationships between independent and dependent variables to predict outcomes in new data instances.

Ensemble techniques, on the other hand, leverage the power of combining multiple models to address challenges in prediction accuracy and optimization. Techniques such as Random Forest (RF), Gradient Boosting (GB), and Adaptive Boosting (AB) are used to build a collection of models and aggregate their outputs, yielding predictions that are more accurate and resilient to overfitting than individual models. These methods are particularly effective in optimizing prediction results, as they combine the strengths of diverse models, thus enhancing the overall performance and stability of the predictive system.

Deep learning (DL) techniques, including architectures like Convolutional Neural Networks (CNN), Long Short-Term Memory networks (LSTM), and Gated Recurrent Units (GRU), are used to capture intricate patterns within the data by leveraging multi-layer neural networks. Unlike traditional ML techniques, DL models have a greater capacity to represent complex, high-dimensional relationships, making them well-suited for tasks requiring high-level feature extraction and temporal pattern recognition. These models learn through deep architectures, enabling them to generalize effectively to unseen data and provide powerful predictions for complex datasets. Overall, the combination of ML, ensemble, and DL approaches in this thesis enables a robust and comprehensive approach to predictive modelling.

Table 2.1 presents an overview of the data analysis techniques utilized in this thesis, categorized into three primary groups: Machine Learning (ML), Ensemble Techniques, and Deep Learning (DL). The Machine Learning category includes traditional algorithms such as Support Vector Machine (SVM), K-Nearest Neighbors (KNN), Naive Bayes, and Multi-Layer Perceptron (MLP), which are widely used for their ability to model relationships between independent and dependent variables.

The Ensemble Techniques category highlights methods like Random Forest (RF), Bagging, Gradient Boosting (GB), Adaptive Boosting (AB), and Extreme Gradient Boosting (XGB). These techniques combine multiple models to enhance prediction accuracy and reduce overfitting.

The Deep Learning category showcases advanced neural network architectures such as Convolutional Neural Networks (CNN), Long Short-Term Memory networks (LSTM), Gated Recurrent Units (GRU), and Multi-Layer Perceptrons (MLP). These models are particularly adept at capturing complex patterns in high-dimensional and sequential data.

This categorization reflects the comprehensive approach taken to leverage various techniques for improved predictive performance and reliability

Table 2.1: Data Analysis Techniques

Category	**Techniques**
Machine Learning (ML)	SVM (Support Vector Machine), KNN (K-Nearest Neighbors), Naive Bayes, MLP (Multi-Layer Perceptron)
Ensemble Techniques	RF (Random Forest), Bagging, GB (Gradient Boosting), AB (Adaptive Boosting), XGB (Extreme Gradient Boosting)
Deep Learning (DL)	CNN (Convolutional Neural Network), LSTM (Long Short-Term Memory), GRU (Gated Recurrent Unit), MLP (Multi-Layer Perceptron)

2.6.1 Support Vector Machine

Support Vector Machine (SVM) is a robust supervised learning algorithm commonly applied to classification, although it can also be adapted for regression. SVM aims to find an optimal decision boundary, called a hyperplane, that effectively separates data points into distinct classes with the maximum possible margin[73, 74]. In SVM, the margin is defined as the distance between the hyperplane and the closest data points from each class. These closest points are known as support vectors, and they play a crucial role in defining the boundary[75]. By maximizing the margin, SVM enhances generalization, making it a powerful classifier, especially in high-dimensional spaces [76].

The algorithm performs best when data is linearly separable, but real-world datasets are often more complex and may not be linearly separable. In such cases, SVM uses kernel functions to map the original feature space into a higher-dimensional space, where a linear separation may become possible[77]. This transformation enables SVM to handle non-linear relationships without explicitly increasing the dimensionality of the input data, making it both powerful and computationally efficient[78].

SVM supports various kernel functions:

- Linear Kernel: Applies a linear transformation, useful when the data is already linearly separable.
- Polynomial Kernel: Maps input features into a polynomial space, allowing SVM to capture polynomial relationships between features.
- Radial Basis Function (RBF) Kernel: The most commonly used kernel in SVM; it maps the data into an infinite-dimensional space, which enables it to model complex decision boundaries.
- Sigmoid Kernel: This kernel functions similarly to a neural network activation function and is typically used in specialized cases.

Through these kernels, SVM can handle complex and non-linear relationships, making it highly adaptable to various types of data.

Key Parameters:

1. Kernel: The choice of kernel function is a critical decision in SVM. Each kernel function (linear, polynomial, RBF, sigmoid) transforms the data in different ways. For instance, the linear kernel is computationally efficient and effective for linearly separable data, while RBF is suited for non-linear data. Selecting the right kernel often requires experimentation and tuning based on the dataset characteristics.

2. C (Regularization Parameter): The regularization parameter, C, controls the trade-off between maximizing the margin and minimizing classification errors. When C is high, the model prioritizes classifying all points correctly, even if it leads to a smaller margin (more complex model). Conversely, a smaller C value allows more misclassifications but promotes a wider margin, leading to a simpler model that may generalize better. Thus, C is essential for balancing model complexity with predictive accuracy.

3. Gamma (in RBF Kernel): Gamma controls the influence of a single training example on the decision boundary. A high gamma value creates a more localized impact, resulting in tighter, more complex decision boundaries around each support vector, which can capture more intricate patterns but may lead to overfitting. A low gamma extends the influence of each point, leading to a smoother decision boundary and enhancing the model's ability to generalize. Gamma is especially important when using the RBF kernel, as it directly impacts the decision surface complexity.

2.6.2 K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) is an instance-based, non-parametric algorithm often used for classification, although it can also be adapted for regression [79]. Unlike most algorithms, KNN does not build an explicit model during training instead, it memorizes the training data points and performs predictions based on their proximity to a test point. This is why KNN is considered a lazy learning algorithmâit doesn't involve training in the traditional sense but makes predictions in real-time by examining the relationships between points in the dataset.[80, 81]

In KNN classification, the algorithm finds the K closest training examples (neighbors) to a new input point and assigns a class label based on the majority class among those neighbors. The distance metric used to calculate proximity significantly impacts the model's performance. Euclidean distance is the most commonly used metric, defined mathematically as:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{2.1}$$

where $\mathbf{x}$ and $\mathbf{y}$ represent two data points, and n is the number of features. This distance metric helps determine which points are closest to the query point.

Key Parameters:

1. K (Number of Neighbors): K represents the number of nearest neighbors considered when making predictions. Choosing an optimal K value is essential; a small K may result in a noisy model that overfits, while a large K could lead to underfitting, as it smooths out local distinctions. Typically, K is determined through cross-validation or using domain knowledge. The optimal K can be found using parameter tuning over a range of values to minimize prediction error.

2. Distance Metric: The distance metric in KNN determines how the ”closeness” of points is calculated. Common metrics include Euclidean Distance, Manhattan Distance, and Minkowski Distance.

2.6.3 Naive Bayes

Naive Bayes is a probabilistic classifier grounded in Bayes' theorem, which provides a mathematical framework for predicting class membership based on prior knowledge[82, 83]. The fundamental principle behind Naive Bayes is to estimate the probability of a class C given a set of features $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$. This is expressed using Bayes' theorem as:

$$P(C \mid \mathbf{X}) = \frac{P(\mathbf{X} \mid C)P(C)}{P(\mathbf{X})} \tag{2.2}$$

where:

- $P(C \mid \mathbf{X})$ is the posterior probability of class C given the features $\mathbf{X}$.
- $P(\mathbf{X} \mid C)$ is the likelihood, or the probability of observing features $\mathbf{X}$ given class C.
- $P(C)$ is the prior probability of class C.
- $P(\mathbf{X})$ is the probability of observing the features $\mathbf{X}$ across all classes (this serves as a normalizing factor).

In practice, Naive Bayes simplifies the computation of $P(\mathbf{X} \mid C)$ by making the naive assumption that the features are conditionally independent given the class label. This means that the presence of one feature does not affect the presence of another feature, which drastically simplifies the likelihood computation:

$$P(\mathbf{X} \mid C) = \prod_{i=1}^{n} P(x_i \mid C) \tag{2.3}$$

As a result, the posterior probability can be rewritten as:

$$P(C \mid \mathbf{X}) \propto P(C) \prod_{i=1}^{n} P(x_i \mid C) \tag{2.4}$$

This formulation allows Naive Bayes to efficiently compute class probabilities for large datasets by leveraging the independence assumption, even if it is a simplification of reality.

To classify a new observation, Naive Bayes computes the posterior probabilities for each class and selects the class C^* that maximizes this probability:

$$C^* = \arg\max_{C} P(C \mid \mathbf{X}) \tag{2.5}$$

Common Types of Naive Bayes:

- Gaussian Naive Bayes: Used when features are continuous and assumed to follow a Gaussian (normal) distribution.
- Multinomial Naive Bayes: Suitable for discrete count data, such as word counts in text classification. It models the likelihood using the multinomial distribution.
- Bernoulli Naive Bayes: Used for binary/boolean features.

2.6.4 Multi Layer Perceptron

A Multi-Layer Perceptron (MLP) is a class of feed-forward artificial neural network that consists of multiple layers of nodes, including an input layer, one or more hidden layers, and an output layer. Each node in the MLP is a neuron that receives input, processes it, and passes the output to the next layer. MLPs are widely used for both classification and regression tasks due to their ability to model complex, non-linear relationships in data.[84–86]

Architecture:

- Input Layer: This layer receives the input features of the dataset. Each node in this layer corresponds to one feature.
- Hidden Layers: These layers consist of multiple neurons (nodes) that perform transformations on the input data. The depth (number of hidden layers) and width (number of neurons in each layer) of the MLP significantly influence its capacity to learn from data.

- Output Layer: The final layer produces the output, which can be a single value (for regression) or a probability distribution across classes (for classification).

Forward Propagation:

In MLP, forward propagation involves the following steps:

- Each input feature is multiplied by its corresponding weight.
- The weighted inputs are summed and passed through an activation function, producing the output for each neuron.

Mathematically, for a neuron j in a layer l, the output $y_j^{(l)}$ can be expressed as:

$$y_j^{(l)} = f\left(\sum_{i=1}^{n} w_{ij}^{(l)} y_i^{(l-1)} + b_j^{(l)}\right) \tag{2.6}$$

where:

- $w_{ij}^{(l)}$ is the weight connecting neuron i from the previous layer $(l-1)$ to neuron j in the current layer l.
- $y_i^{(l-1)}$ is the output of neuron i from the previous layer.
- $b_j^{(l)}$ is the bias term for neuron j in layer l.
- f is the activation function applied to the weighted sum.

Backpropagation:

The backpropagation algorithm is used to minimize the error between the predicted output and the actual target value. It updates the weights of the network based on the gradient of the loss function with respect to each weight. The loss function can be defined using various metrics, such as Mean Squared Error (MSE) for regression tasks or Cross-Entropy Loss for classification tasks. For instance, the MSE is given by:

$$L = \frac{1}{N} \sum_{k=1}^{N} (y_k - \hat{y}_k)^2 \tag{2.7}$$

where:

- L is the loss,
- N is the number of samples,
- y_k is the true value, and
- $\hat{y}_k$ is the predicted value.

During backpropagation, the gradients of the loss with respect to the weights are calculated using the chain rule. The weight update rule can be expressed as:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial L}{\partial w_{ij}^{(l)}} \tag{2.8}$$

where:

- η is the learning rate, controlling the size of the weight updates.

Key Parameters:

- Hidden Layers and Units:Determines the network's capacity to model complex relationships.
- Activation Function: Defines the non-linear transformation applied to the weighted input.
- Learning Rate: Controls the size of updates during training.
- Batch Size: The number of samples processed before updating weights.
- Epochs: The number of passes through the entire training dataset.

2.6.5 Random Forest (RF)

Random Forest (RF) is an ensemble learning method that constructs a multitude of decision trees during training and outputs the class that is the mode (for classification) or the average (for regression) of the classes (or values) predicted by individual trees[87–90]. The primary motivation behind using an ensemble of trees is to improve predictive accuracy and control overfitting, which is a common issue in decision trees due to their high variance.

The process of creating a Random Forest can be broken down into several key steps:

1. Bootstrapping: For each tree, a random sample of the data (with replacement) is drawn. This is known as a bootstrapped sample. As a result, some observations will be repeated, while others may not be included in the sample at all. This randomness in the training set helps in reducing overfitting.

2. Building Decision Trees: Each decision tree is constructed using the bootstrapped sample. When building a tree, at each node, a random subset of features is selected (rather than considering all features). This randomness leads to trees that are decorrelated, meaning they make different errors on the training data.

Mathematically, the prediction of a Random Forest can be represented as:

$$\hat{y} = \frac{1}{N} \sum_{i=1}^{N} T_i(x) \tag{2.9}$$

where:

- $\hat{y}$ is the predicted value,

- N is the number of trees in the forest,
- $T_i(x)$ is the prediction made by the i-th decision tree for input x.

Aggregation of Predictions:

- For classification tasks, the final prediction is based on majority voting, where each tree votes for its predicted class, and the class with the most votes is selected as the final output.
- For regression tasks, the average of all tree predictions is calculated.

The mode for classification can be expressed as:

$$\hat{y} = \text{mode}\{T_1(x), T_2(x), \ldots, T_N(x)\} \tag{2.10}$$

Key Parameters:

- Number of Trees: Controls the size of the ensemble. A larger number of trees generally improves stability but increases computational cost.
- Max Depth: Specifies the maximum depth of each tree. Deeper trees capture more complex patterns but may overfit the training data.
- Max Features: Limits the number of features considered for splitting at each node, introducing randomness and reducing overfitting.
- Min Samples Split: Determines the minimum number of samples required to split an internal node. Higher values can reduce overfitting by limiting tree depth.

2.6.6 Bootstrap Aggregating (Bagging)

Bagging, short for Bootstrap Aggregating, is an ensemble learning technique designed to improve the stability and accuracy of machine learning algorithms. The primary goal of Bagging is to reduce the variance of a predictive model by training multiple instances of the same algorithm on different subsets of the training dataset, allowing for the averaging of predictions [81, 91, 92]

The process begins by creating several bootstrapped samples from the original dataset. A bootstrapped sample is formed by randomly selecting data points with replacement, meaning the same data point can appear multiple times in the same sample. The size of each bootstrapped sample is usually the same as the original dataset.For each bootstrapped sample, a separate model (base estimator) is trained. By combining the predictions of multiple models, Bagging reduces the likelihood of overfitting and enhances the generalization capability of the ensemble.

2.6.7 Gradient Boosting (GB)

Gradient Boosting is an ensemble learning technique that constructs a predictive model in a sequential manner, focusing on minimizing a specified loss function through iterative updates. The primary goal of Gradient Boosting is to build a strong predictive model by combining multiple weak learners, typically decision trees, into a single, robust ensemble model.[93, 94]

The process begins with an initial prediction, often the mean of the target values for regression tasks. Each subsequent tree is trained to predict the residuals (errors) of the predictions made by the previous trees. This method allows the model to learn from mistakes, effectively correcting the errors of its predecessors.[95]

Mathematically, if $F(x)$ represents the current prediction for an instance x, and y

is the true target value, the residual error can be defined as:

$$r_i = y_i - F(x_i) \tag{2.11}$$

where r_i is the residual for the i-th instance.

The next tree $h_m(x)$ is fit to the residuals, and the model is updated as follows:

$$F_m(x) = F_{m-1}(x) + \eta \cdot h_m(x) \tag{2.12}$$

where:

- $F_{m-1}(x)$ is the prediction from the previous iteration,
- η is the learning rate that controls how much of the new tree's prediction contributes to the overall model,
- $h_m(x)$ is the new tree being added.

The learning rate is a crucial parameter as it determines how quickly the model adapts to the training data. A smaller learning rate typically requires more boosting iterations to achieve optimal performance, while a larger learning rate can lead to overfitting.

Gradient Boosting uses gradient descent to minimize the loss function, which could be mean squared error (MSE) for regression tasks or log loss for classification tasks. The update process can be visualized through the following objective function that needs to be minimized:

$$L(F) = \sum_{i=1}^{N} l(y_i, F(x_i)) + \Omega(F) \tag{2.13}$$

where:

- $L(F)$ is the total loss,
- $l(y_i, F(x_i))$ is the loss function (e.g., squared error for regression),
- $\Omega(F)$ is a regularization term to prevent overfitting.

By iteratively reducing the residual errors and minimizing the loss function, Gradient Boosting builds a powerful and robust predictive model capable of handling both regression and classification tasks.

2.6.8 Adaptive Boosting (AdaBoost)

Adaptive Boosting, commonly referred to as AdaBoost, is a powerful ensemble learning technique that enhances the performance of weak classifiers by combining their outputs into a single strong learner. The fundamental idea behind AdaBoost is to iteratively train multiple weak learners, typically decision stumps (single-level decision trees), where each learner is trained to correct the errors made by the previous ones.[96, 97]

The process begins with assigning equal weights to all training samples. In each iteration t, the algorithm focuses on the samples that were misclassified by the previous weak learner. Misclassified samples are assigned higher weights, forcing the subsequent weak learner to pay more attention to them. The steps involved in the AdaBoost algorithm can be summarized as follows:

1. Initialization: Set the initial weights of all training samples:

$$w_i = \frac{1}{N}, \quad \text{for } i = 1, 2, \ldots, N \tag{2.14}$$

where N is the number of training samples.

2. For each iteration $t = 1, 2, \ldots, T$:

   (a) Train a weak classifier $h_t(x)$ using the weighted training data.

   (b) Calculate the error ϵ_t of the weak classifier:

   $$\epsilon_t = \frac{\sum_{i=1}^{N} w_i \cdot I(y_i \neq h_t(x_i))}{\sum_{i=1}^{N} w_i} \tag{2.15}$$

   where I is the indicator function that equals 1 if the sample is misclassified and 0 otherwise.

   (c) Compute the classifier's weight α_t based on its performance:

   $$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right) \tag{2.16}$$

   (d) Update the weights of the training samples:

   $$w_i \leftarrow w_i \cdot \exp\left(-\alpha_t \cdot y_i \cdot h_t(x_i)\right) \tag{2.17}$$

   where y_i is the true label of the i-th sample. This adjustment increases the weights for misclassified samples and decreases the weights for correctly classified ones.

   (e) Normalize the weights:

   $$w_i \leftarrow \frac{w_i}{\sum_{i=1}^{N} w_i} \tag{2.18}$$

3. Final Model: The final strong classifier $H(x)$ is a weighted sum of the individual weak classifiers:

   $$H(x) = \sum_{t=1}^{T} \alpha_t \cdot h_t(x) \tag{2.19}$$

The prediction is made by applying a sign function to $H(x)$:

$$\text{Prediction}(x) = \text{sign}(H(x)) \tag{2.20}$$

Through this iterative correction process, AdaBoost effectively converts weak learners into a robust composite model, significantly improving predictive accuracy.

2.6.9 XGBoost (Extreme Gradient Boosting)

XGBoost (Extreme Gradient Boosting) is an advanced implementation of the gradient boosting framework, designed for efficiency, flexibility, and portability. It improves upon traditional gradient boosting by integrating optimization techniques and regularization to enhance model performance and prevent overfitting.[98]

In XGBoost, trees are constructed sequentially, meaning each new tree aims to correct the errors made by the previously built trees. This approach focuses on minimizing a custom loss function, typically based on the residuals of the predictions, allowing the model to learn from its mistakes.[22]

The key concept behind XGBoost is the use of gradient descent to minimize the loss function. For a given dataset with n samples and m features, XGBoost can be represented mathematically as follows:

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \alpha_t \cdot h_t(x_i) \tag{2.21}$$

where:

- $\hat{y}_i^{(t)}$ is the prediction for the i-th instance after t iterations,
- $\hat{y}_i^{(t-1)}$ is the prediction for the i-th instance from the previous iteration,
- α_t is the learning rate (shrinkage),

- $h_t(x_i)$ is the output of the new tree added at iteration t.

The loss function L in XGBoost can be expressed as:

$$L = \sum_{i=1}^{n} \ell(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{2.22}$$

where:

- $\ell(y_i, \hat{y}_i)$ is the loss function measuring the difference between the actual and predicted values for sample i,
- $\Omega(f_k)$ is a regularization term for the k-th tree.

The use of regularization (both L1 and L2) helps to control model complexity and avoid overfitting, making XGBoost particularly powerful for real-world applications where datasets can be noisy or sparse.

2.6.10 CNN (Convolutional Neural Networks)

Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed to process and analyze spatial data, particularly images. CNNs excel at automatically detecting patterns and features in images by mimicking the human visual system. The architecture of CNNs typically consists of several layers, including convolutional layers, pooling layers, and fully connected layers, which work together to transform raw pixel data into meaningful representations for tasks such as image classification, object detection, and segmentation.[99, 100]

1. Convolutional Layers: The core component of a CNN is the convolutional layer, which applies a set of learnable filters (also known as kernels) to the input image. Each filter detects specific features, such as edges or textures, by sliding across the image and performing a dot product operation between the filter and

the input data. Mathematically, the output O of a convolutional layer can be represented as:

$$O(i,j) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} I(i+m, j+n) \cdot K(m,n) \tag{2.23}$$

where:

- I is the input image,
- K is the filter or kernel,
- (i,j) are the spatial dimensions of the output feature map,
- k is half the size of the filter dimensions.

This operation highlights the features present in the image, enabling the network to learn hierarchical representations as the data passes through multiple convolutional layers.

2. Activation Function: After convolution, an activation function, usually ReLU (Rectified Linear Unit), is applied to introduce non-linearity into the model. The ReLU function is defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{2.24}$$

This step allows the network to learn complex patterns.

3. Pooling Layers: Following convolutional layers, pooling layers reduce the spatial dimensions of the feature maps while retaining the most critical information. This helps decrease the number of parameters and computation in the network, mitigating the risk of overfitting. The most common pooling method is max

pooling, defined as:

$$O(i,j) = \max_{m,n}\{I(s \cdot i + m, s \cdot j + n)\} \tag{2.25}$$

where:

- s is the stride, indicating how far the pooling window moves,
- m and n span the dimensions of the pooling window.

Pooling reduces the dimensionality while preserving the features extracted by the convolutional layers.

4. Fully Connected Layers: After several convolutional and pooling layers, the final feature maps are flattened and fed into one or more fully connected layers. Each neuron in these layers is connected to every neuron in the previous layer, enabling the network to make final predictions based on the learned features. The output can be computed as:

$$O = \sigma(W \cdot X + b) \tag{2.26}$$

where:

- W is the weight matrix,
- X is the input from the previous layer,
- b is the bias term,
- σ is the activation function, commonly softmax for multi-class classification.

2.6.11 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a specialized type of recurrent neural network (RNN) explicitly designed to capture long-range dependencies and avoid the vanishing gradient problem that traditional RNNs often face. LSTMs achieve this by utilizing memory cells and gates that regulate the flow of information.[101–103]

An LSTM cell consists of three primary gates:

1. Input Gate: Controls the extent to which new information flows into the cell.
2. Forget Gate: Decides what information to discard from the cell state.
3. Output Gate: Determines what information to output from the cell.

The key equations governing these operations are as follows:

1. Forget Gate:
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.27}$$
   where:
   - f_t is the forget gate vector,
   - σ is the sigmoid activation function,
   - W_f is the weight matrix for the forget gate,
   - h_{t-1} is the hidden state from the previous time step,
   - x_t is the current input,
   - b_f is the bias.
2. Input Gate:
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i), \quad \tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{2.28}$$

where:

- i_t is the input gate vector,
- $\tilde{C}_t$ is the candidate cell state vector,
- W_i and W_C are the weight matrices for the input gate and candidate state,
- b_i and b_C are the biases.

3. Cell State Update:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{2.29}$$

where:

- C_t is the updated cell state,
- C_{t-1} is the previous cell state.

4. Output Gate:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o), \quad h_t = o_t \cdot \tanh(C_t) \tag{2.30}$$

where:

- o_t is the output gate vector,
- h_t is the hidden state,
- W_o is the weight matrix for the output gate,
- b_o is the bias.

2.6.12 Gated Recurrent Unit

Gated Recurrent Units (GRUs) are a simplified variant of LSTMs, designed to retain the capability of capturing long-term dependencies while being more computationally efficient. GRUs combine the forget and input gates into a single update gate and have a reset gate that controls the influence of the previous hidden state.[100, 104]

The equations for a GRU are as follows:

1. Update Gate:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \tag{2.31}$$

   where:

   - z_t is the update gate vector,
   - σ is the sigmoid activation function,
   - W_z is the weight matrix for the update gate,
   - h_{t-1} is the previous hidden state,
   - x_t is the current input,
   - b_z is the bias term.

2. Reset Gate:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \tag{2.32}$$

   where:

   - r_t is the reset gate vector,
   - W_r is the weight matrix for the reset gate,
   - b_r is the bias term.

3. Candidate Activation:

$$\tilde{h}_t = \tanh(W_h \cdot [r_t \cdot h_{t-1}, x_t] + b_h) \tag{2.33}$$

   where:

- $\tilde{h}_t$ is the candidate activation,
- W_h is the weight matrix for the candidate activation,
- b_h is the bias term.

4. Final Hidden State:

$$h_t = z_t \cdot h_{t-1} + (1 - z_t) \cdot \tilde{h}_t \tag{2.34}$$

where:

- h_t is the final hidden state,
- z_t is the update gate vector,
- $\tilde{h}_t$ is the candidate activation.

The final hidden state h_t is a linear interpolation between the previous hidden state h_{t-1} and the candidate activation $\tilde{h}_t$, controlled by the update gate z_t.

2.7 Empirical Data Collection

Empirical validation data, which is essential for assessing and improving software prediction models, can be drawn from various sources, including industrial software systems, open-source projects, and academic or research-driven software systems. Each source type offers distinct advantages and limitations in terms of accessibility, cost, and relevance. Recently, however, open-source datasets have emerged as the predominant choice for empirical validation, largely due to their inherent benefits and the support of a collaborative community that continues to enhance their utility.

One of the primary reasons for the increasing popularity of open-source datasets in empirical validation is ease of accessibility. Open-source projects are typically available to the public, allowing researchers to access a wide array of data without the limitations posed by proprietary or restricted systems. Additionally, open-source datasets are cost-effective as they often eliminate licensing fees or usage restrictions, making them accessible for researchers or institutions with limited resources.

In light of these advantages, this thesis primarily utilizes various open-source datasets to empirically validate the proposed models. These datasets, chosen for their rich characteristics and broad applicability, allow for in-depth analysis and benchmarking of software defect prediction methods. To establish a systematic approach, the thesis begins by outlining the data collection process, detailing how these datasets were acquired, pre-processed, and prepared for analysis. Following this, each dataset is described in detail, with an emphasis on key attributes such as defect rates, project size, and code complexity that are crucial for understanding their role in empirical validation within the study.

This thesis utilizes a range of open-source software datasets from diverse domains. The main goal of selecting datasets across different domains was to achieve more generalized results. Additionally, the open-source nature of these datasets enhances the replicability of the findings.

2.7.1 Dataset Details

In the early stages, some academic researchers and companies employed non-public datasets, such as proprietary projects, to develop defect prediction models [105]. However, it is not possible to compare the results of such methods to each other, as these datasets cannot be obtained. Since machine learning researchers faced similar problems in the 1990s, they created a repository called the University of California Irvine (UCI) Machine Learning Repository. Inspired by the success of the UCI repository, researchers created the PROMISE repository of empirical software engineering data, which has collected several publicly available datasets since 2005.

In addition, some researchers spontaneously publish their extracted datasets for further empirical study on software defect prediction.

In this section, we briefly introduce existing publicly available and commonly used benchmark datasets.

Table 2.2: Summary of Publicly Available Software Defect Datasets

Dataset	**Number of Projects**	**Metric Used**	**Number of Metrics**	**Granularity**	**Description**
AEEEM	5	CMs, process metrics	61	Class	Open-source projects for evaluating defect prediction models. *http://bug.inf.usi.ch*
NASA	13	CMs	20 to 40	Function	NASA Metrics Data Program. *http://openscience.us/repo/*
PROMISE	38	CMs	20	Class	Open-source, proprietary, and academic projects. *http://openscience.us/repo/*
ReLink	3	CMs	26	File	Recovering links between bugs and changes. *www.cse.ust.hk/scc/ReLink.html*

NASA Dataset

NASA benchmark dataset consists of 13 software projects [36]. The number of instances ranges from 127 to 17,001, while the number of metrics ranges from 20 to 40. Each project in the NASA dataset represents a NASA software system or sub-system, containing the corresponding defect-marking data and various static code metrics (CMs). The repository records the number of defects for each instance using a bug tracking system. The static CMs in the NASA datasets include size, readability, complexity attributes, and other factors closely related to software quality. Each project in NASA dataset represents a NASA software system or sub-system (from various domains such as spacecraft instruments, real time predictive ground system, zero-gravity experiment related to combustion and flight software for earth orbitting satellites).

The Table 2.3 summarizes key details of the NASA datasets, including the pro-

Table 2.3: Summary of NASA Datasets

Dataset	Language	#Modules	#Defective Modules	#Features
CM1	C	498	49	37
JM1	C	7782	1672	21
MW1	C	403	61	37
PC1	C	1107	76	37
PC3	C	1077	134	37
PC4	C	1458	178	37
PC5	C++	1711	471	38

gramming languages used, the number of modules, the count of defective modules, and the number of features for each dataset. These datasets have been extensively used for software defect prediction studies, demonstrating their relevance in empirical validation.

PROMISE Dataset

Jureczko and Madeyski [106] collected some open source, proprietary and academic software projects, which are part of the PROMISE repository as shown in Table 2.4 . The collected data consists of 92 versions of 38 different software development projects, including 48 versions of 15 open-source projects, 27 versions of six proprietary projects and 17 academic projects. Each project has 20 metrics in total which contains McCabe's cyclomatic metrics, CK metrics and other OO metrics.

RELINK Dataset

Three datasets in ReLink were collected by Wu et al. [52] to improve the defect prediction performance by increasing the quality of the defect data. The defect information in ReLink has been manually verified and corrected. Similarly on the basis of link between bugs and changes, Kim et al. used the two projects in Eclipse 3.4. They collected the defect data by mining the Eclipse Bugzilla and CVS repositories and found that both projects have a high percentage of linked bugs (bugs whose changes logs and bug reports are linked)[50]. For SWT, 92.27% bugs reported in Bugzilla are linked to changes. For Debug, 95.92% bugs are linked. The detailed information is prenet in Table 2.5

Table 2.4: Summary of PROMISE Repository Datasets Investigated by Jureczko and Madeyski

Description and Source	Dataset	Language	#Modules	#Defective Modules	#Features
Different releases of open-source projects	Ant-1.7	Java	745	166	20
	Camel-1.2	Java	608	216	20
	Camel-1.4	Java	872	145	20
	Camel-1.6	Java	965	188	20
	Derby-10.2.1.6	Java	1963	648	20
	Derby-10.3.1.4	Java	2206	661	20
	Ivy-2.0	Java	352	40	20
	Jedit-3.2.1	Java	272	90	20
	Jedit-4.1	Java	312	79	20
	Jedit-4.2	Java	367	48	20
	Jedit-4.3	Java	492	11	20
	Log4j-1.0	Java	135	34	20
	Log4j-1.1	Java	109	37	20
	Log4j-1.2	Java	205	189	20
	Lucene-2	Java	195	91	20
	Lucene-2.2	Java	247	144	20
	Poi-3.0	Java	442	281	20
	Tomcat-6.0	Java	858	77	20
	Xalan-2.4	Java	723	110	20
	Xalan-2.5	Java	803	387	20
	Xalan-2.6	Java	885	411	20
	Xalan-2.7	Java	909	898	20
	Xerces-1.4	Java	588	437	20
Academic software	Arc	Java	234	27	20
Proprietary software projects investigated by Jureczko et al.	Prop-1	Java	18471	2738	20
	Prop-2	Java	23014	2431	20
	Prop-3	Java	10274	1180	20
	Prop-4	Java	8718	840	20
	Prop-5	Java	8516	1299	20
	Prop-6	Java	660	66	20

Table 2.5: Summary of Additional Software Datasets

Dataset	Language	#Modules	#Defective Modules	#Features
Apache	Java	194	98	26
Safe (OpenIntents)	Java	56	22	26
Zxing	Java	399	118	26
Debug	Java	1065	263	17
SWT	Java	1485	653	17

AEEEM dataset

AEEEM was used to benchmark different defect prediction models and collected by D'Ambros et al. [2] Each AEEEM dataset consists of 61 metrics: 17 source CMs, five previous-defect metrics, five entropy-of-change metrics, 17 entropy-of-source CMs, and 17 churn-of-source CMs [16].

Table 2.6: Summary of Software Datasets

Dataset	Language	#Modules	#Defective Modules	#Features
EQ	Java	324	129	61
JDT	Java	997	206	61
LC	Java	691	64	61
ML	Java	1862	245	61
PDE	Java	1497	209	61

2.8 Data Preprocessing

Data preprocessing is essential in software defect prediction to improve data quality and model accuracy. First, datasets are carefully examined for redundant data, such as duplicate rows or columns, as these can skew model predictions. Missing values are handled through imputation techniques (e.g., filling with mean or median values) or by removing entries with excessive gaps, thereby reducing biases. Next, normalization scales feature consistently, using either Min-Max normalizationâscaling values between 0 and 1 while preserving relationshipsâor standard scaling, which centres data with a mean of zero and unit variance, ensuring no feature dominates due to scale. Feature selection is then performed to retain only the most relevant attributes, enhancing model efficiency and interpretability by reducing dimensionality and removing irrelevant or noisy features. Additionally, data balancing techniques, such as oversampling minority classes, are employed to ensure that defect and non-defect classes are represented fairly, preventing the model from favouring one class over the other. Together, these preprocessing steps refine the dataset, enabling more accurate, reliable, and generalizable defect predictions essential for robust software engineering predictive modelling.

2.8.1 Data Normalisation

Data normalization is a crucial preprocessing step in software defect prediction, as it enhances model performance and accuracy by standardizing feature scales.[107] The primary goals of normalization in this context are to bring features to a common scale, thereby preventing features with larger magnitudes from dominating the model, improving the convergence rate of algorithms, and reducing the impact of outliers.[55, 108] Common techniques include Min-Max scaling, and Z-score normalization, which centres data with a mean of 0 and a standard deviation of 1, and log transformation, which compresses wide-ranging values. Normalization yields several benefits, such as improved model accuracy, faster convergence for iterative algorithms, and more meaningful comparisons between software metrics. To maximize these benefits, practitioners must select normalization techniques based on data distribution and ensure consistent application across training and testing datasets. Properly normalized data enhances defect prediction, enabling accurate identification of potential software defects and optimizing testing resource allocation.

Min-Max Scaling

Min-Max scaling transforms the data to a fixed range, typically between 0 and 1 [100]. The formula for Min-Max scaling is as follows:

$$x_{\text{scaled}} = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \tag{2.35}$$

Where:

- x is the original value,
- x_{min} is the minimum value of the feature,
- x_{max} is the maximum value of the feature.

This technique is useful when you want to preserve zero values in sparse data and when the distribution of the data is not Gaussian or unknown.

Z-score Normalization (Standard Scaling)

Z-score normalization transforms the data to have a mean of 0 and a standard deviation of 1 [80]. The formula for Z-score normalization is as follows:

$$x_{\text{scaled}} = \frac{x - \mu}{\sigma} \tag{2.36}$$

Where:

- x is the original value,
- μ is the mean of the feature,
- σ is the standard deviation of the feature.

This technique is particularly useful when the data follows a Gaussian distribution and when dealing with outliers.

2.9 Feature Reduction

According to ”curse of dimensionality” as the number of dimensions (features) increases, the volume of the space increases exponentially, making the available data sparse. This sparsity can make it difficult to obtain reliable and accurate models. Feature reduction techniques can be categorized into feature selection and feature extraction. Feature selection is further subdivided into filter, wrapper, and embedded models. The hierarchical figure 2.2 emphasizes the systematic approach to reducing high-dimensional data for efficient analysis, enhancing computational performance, and improving model accuracy in various machine learning applications. Feature Selection which involves selecting a subset of the most relevant features from the original dataset without transforming them. The goal is to improve model performance by reducing overfitting and enhancing interpretability. .[Z. M. Hira and D. F. Gillies, âA Review of Feature Selection and Feature Extraction Methods Applied on Microarray Dataâ, Advances in Bioinformatics, vol.4, 2015.]. Feature Extraction involves transforming the original features into a new set of features that capture the most important information. This typically results in a lower-dimensional space, where the new features may be combinations of the original features. [Z. A. Rana, M. M. Awais and S. Shamail, âImpact of Using Information Gain in Software Defect Prediction Modelsâ, Intelligent Computing Theory 2014, pp. 637â648, 2014.]

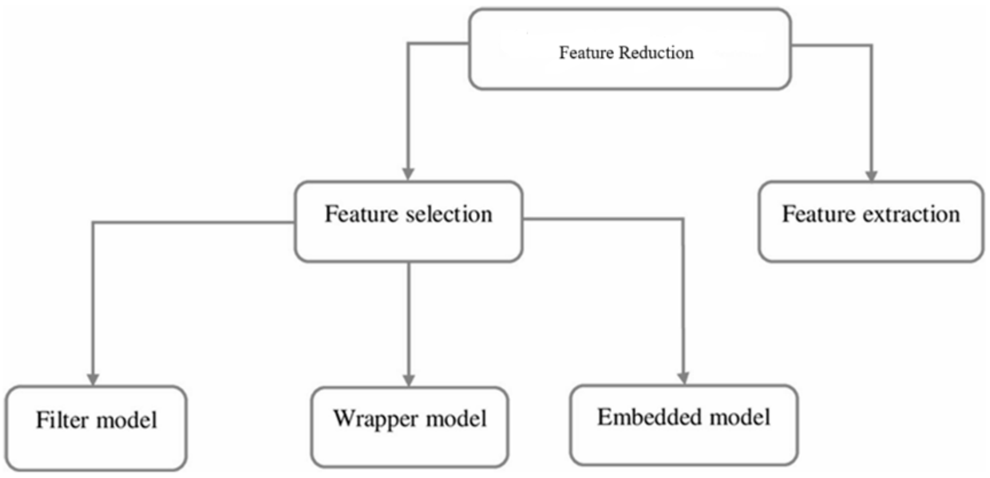


Figure 2.2: Types of Feature Reduction Technique

2.9.1 Feature Extraction

Feature extraction is an essential technique in machine learning and data analysis that focuses on identifying and extracting relevant features from raw data to create a more informative dataset as shown in 2.3. Its primary purposes are to reduce data complexity (dimensionality) while retaining important information, enhance the performance and efficiency of machine learning algorithms, and simplify the analysis process. Common feature extraction techniques include Principal Component Analysis (PCA), which reduces dimensionality while preserving maximum variance and uncovering patterns between variables; Linear Discriminant Analysis (LDA), which, like PCA, reduces dimensions but also considers class labels to maximize class separability; and autoencoders, a neural network-based approach that learns compressed representations of data by training the network to recreate its input, thereby discovering underlying data structures. The benefits of feature extraction include improved model accuracy and efficiency, reduced redundancy, faster learning speeds, and enhanced interpretability of models.

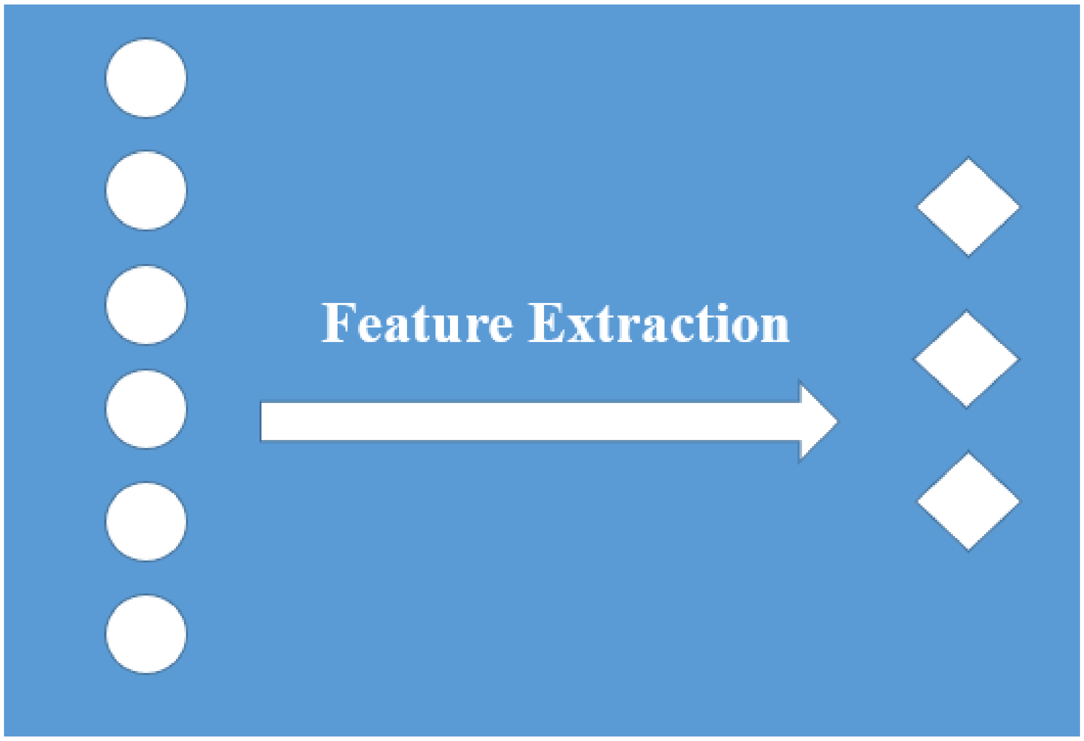


Figure 2.3: Types of Feature Reduction Technique

Types of Feature Extraction Techniques

In this thesis, various feature extraction techniques were employed to enhance the model's ability to predict software defects effectively. The techniques used include

Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA), Kernel-based Principal Component Analysis (K-PCA), and Autoencoders. Each of these methods serves a unique purpose in transforming the original dataset into a more manageable form, which retains the most significant information for analysis.

1. **Principal Component Analysis (PCA)**

   PCA is a widely used technique that identifies principal components, which are linear combinations of the original features that are orthogonal to each other. The goal of PCA is to capture the maximum variance in the data with the least number of principal components. Typically, this means that only a small number of principal components are necessary to explain a significant portion of the data's variability.[109] To find these principal components, the covariance matrix of the original data is computed, and the eigenvalues of this matrix are determined. The principal components are then represented by the eigenvectors associated with the largest eigenvalues.This dimensionality reduction technique helps simplify the dataset while preserving its essential characteristics, making it easier for predictive models to learn from the data.

2. **Kernel-PCA (K-PCA)**

   K-PCA extends the traditional PCA into high-dimensional spaces through the use of kernel functions. This approach is particularly useful for handling non-linear relationships within the data.The kernel trick allows K-PCA to operate in a transformed feature space without explicitly computing the coordinates of the data in that space. Instead of using the covariance matrix, K-PCA calculates the principal eigenvectors from the kernel matrix.This method has gained popularity due to its effectiveness in various applications, including those involving support vector machines, as it can reveal structure in the data that linear methods might miss [110, 111].

3. **Linear Discriminant Analysis (LDA)**

   LDA focuses on maximizing the separability between different classes within the dataset. The main objective of LDA is to find a set of vectors that provide the maximum separation between classes while minimizing the variance within each class. This is achieved by analyzing the scatter matrices that represent the variability within and between classes. By emphasizing the differences between classes, LDA enhances the model's ability to classify new data points effectively. This technique is particularly valuable in scenarios where the goal is to discriminate among multiple classes, making it relevant for software defect prediction [112].

4. **Autoencoders**

   Autoencoders are a type of neural network used for feature extraction and dimensionality reduction. This technique allows for the learning of a non-linear mapping between high-dimensional input data and a lower-dimensional representation. A key advantage of autoencoders is their ability to stack multiple hidden layers, which enables the generation of various levels of abstract features. Typically, sigmoid activation functions are employed to facilitate the non-linear transformations within the network.Autoencoders learn to compress the input data into a compact representation and then reconstruct it, effectively capturing the essential patterns within the data. This makes them a powerful tool for feature extraction, particularly in complex datasets where traditional linear methods may fall short [113].

2.9.2 Feature selection

Feature selection is a key preprocessing step in software defect prediction, aimed at identifying the most relevant features to improve model outcomes. Its primary purposes are to reduce data dimensionality, eliminate irrelevant or redundant features, enhance model accuracy, reduce overfitting, and make models more interpretable. Feature selection is a process of selecting a subset of relevant features from the original set of features as shown in Figure 2.4. Feature selection benefits defect prediction by boosting model accuracy, reducing computational demands, and identifying key factors contributing to software defects. Challenges include computational costs of finding optimal feature subsets, variability across selection methods, and balancing noise removal with information retention. Recent research suggests that hybrid approaches, such as combining filter and wrapper methods, as well as multi-objective techniques, can optimize feature selection for both accuracy and efficiency.[114] Effective feature selection addresses high-dimensionality issues in software metrics, significantly enhancing prediction performance when tailored to dataset and model requirements and validated for generalization.

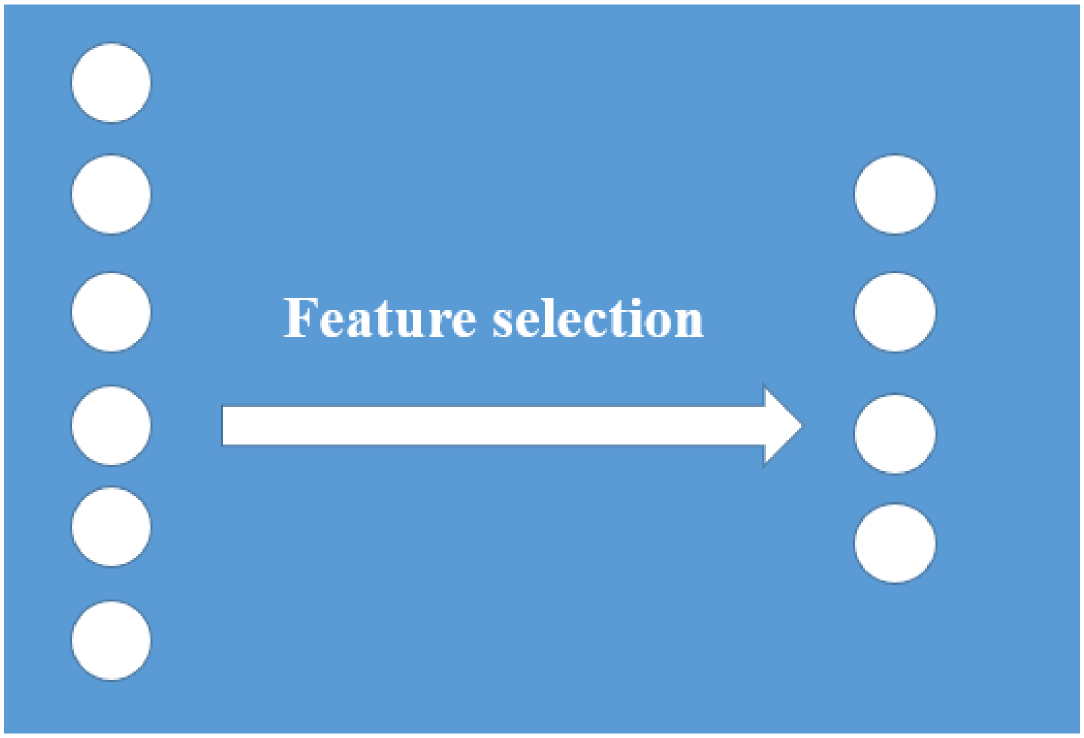


Figure 2.4: Types of Feature Reduction Technique

Types of Feature Selection Techniques

Common feature reduction techniques include filter methods, which use statistical measures to select features independently of the model (e.g., correlation, chi-squared, information gain); wrapper methods, which evaluate feature subsets using predictive models (e.g., recursive feature elimination); and embedded methods, which perform feature selection during model training (e.g., LASSO and decision tree importance). In this thesis following techniques are utilized to perform feature selection.

1. **Filter Methods**

   Filter methods are a feature selection technique that evaluates features based on statistical criteria before model training, making them computationally efficient and highly scalable for high-dimensional data. Independent of any specific learning algorithm, filter methods use statistical measures to score each feature, rank them accordingly, and then select the top K features or those above a defined threshold as shown in Figure 2.5. This flexibility allows them to be used with various machine learning models, and they are often applied as an initial step to swiftly remove irrelevant or redundant features, especially in large datasets, streamlining the data for more effective model training.

Types of filter techniques

In this thesis, various feature selection techniques were employed to enhance the performance and accuracy of software defect prediction models. Methods such as Gain Ratio, Symmetric Uncertainty, ReliefF, Information Gain, and OneR were utilized to evaluate and rank features based on their relevance and ability to distinguish between defect-prone and non-defect-prone instances.

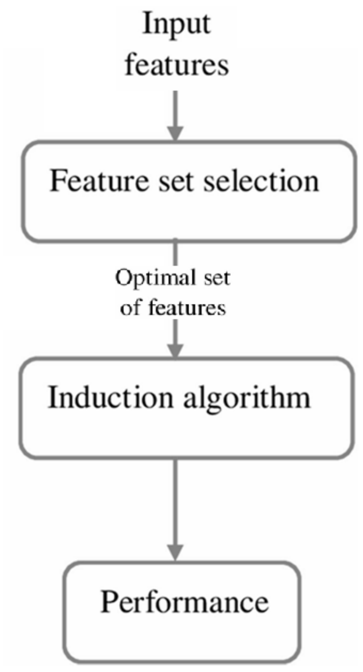


Figure 2.5: Working of Filter Technique

(a) **Gain Ratio (GR):** Gain Ratio is a modified type of information gain that penalizes multivalued attributes to minimize bias. It is defined as the ratio of Information Gain to the intrinsic information of the attribute:

$$GR(T, a) = \frac{\text{Information Gain}(T, a)}{\text{Intrinsic Information}(a)} \tag{2.37}$$

Where T is the target variable and a is the attribute being evaluated [115, 116]

(b) **Symmetrical Uncertainty (SU):** It identifies the dependency between two features. SU is used to measure the relevance of a feature to the target class. Symmetric Uncertainty is defined as:

$$SU(X, Y) = 2 \cdot \frac{I(X; Y)}{H(X) + H(Y)} \tag{2.38}$$

Where $I(X; Y)$ is the mutual information, and $H(X)$, $H(Y)$ are the entropies of X and Y, respectively [117].

(c) **OneR (One Rule):** This method generates one rule for each predictor in the dataset, then selects the rule with the smallest total error as its ”one rule.” OneR is a simple and interpretable feature selection method that evaluates each feature in a dataset individually to determine its predictive power for a target variable. The process begins by creating a frequency table for each feature against the target variable, where each unique value of the feature is paired with the most frequent class label. This establishes a ”rule” for predicting the target variable based on that feature alone. The error rate of this rule is then calculated by measuring how often it incorrectly classifies instances. This process is repeated for all features, and the feature with the lowest error rate is selected as the most important, as it provides the most accurate prediction of the target variable. OneR's simplicity and transparency make it useful for initial feature selection, though it may not capture complex relationships among features.[118]

(d) **Information Gain (IG):** Information Gain is a popular feature selection technique that measures how much information a feature provides about the target variable. The IG approach is based on entropy. After examining the attributes, IG calculates the decrease in uncertainty about the class tag.[119] IG's bias is that it prefers to pick features of higher values.

(e) **ReliefF:** The value of a feature is determined by sampling an object repeatedly and evaluating the magnitude of the given attribute for the closest instances of the same and different classes.[120]

Working Steps of the ReliefF Algorithm

i. Random Sampling: ReliefF begins by randomly selecting an instance from the dataset, referred to as the sample instance.

ii. Finding Nearest Neighbors:

- Nearest Hits: Instances that belong to the same class as the sample.
- Nearest Misses: Instances from each of the other classes that are closest to the sample instance.

This ensures that the algorithm considers both within-class and between-class relationships for each feature.

iii. Feature Weight Update:

- For each feature, ReliefF updates a weight based on how well the feature differentiates between the sample instance and its neighbors.

- Specifically, the algorithm increases the weight if the feature value differs between the sample and its nearest miss (indicating that it can separate different classes).
- Conversely, the algorithm decreases the weight if the feature value differs between the sample and its nearest hit (indicating inconsistency within the same class).
- Mathematically, for a feature f, the weight update rule can be expressed as:

$$W[f] = W[f] + \frac{1}{m}\sum_{i=1}^{m}(\Delta_{\text{Miss}} - \Delta_{\text{Hit}})$$

where m is the number of sampled instances, and the differences depend on the feature values between the sample and its neighbors.

iv. Repeating the Process:

- Steps 1-3 are repeated for a predefined number of sampled instances.
- The algorithm can revisit features multiple times, refining the weights as it processes more samples.

v. Ranking and Selecting Features:

- After iterating through multiple samples, the algorithm produces a weight for each feature.
- Higher weights indicate more relevant features, as they consistently contribute to distinguishing between classes.
- Features with the highest weights are selected for the final model.

2. **Wrapper Methods**

Wrapper feature selection is a powerful technique for identifying the most relevant features for classification tasks, as it tailors feature subsets to a specific classification model by ”wrapping” the selection process around the algorithm as shown in Figure . It involves generating different feature subsets, training a classifier on each subset, evaluating its performance, and selecting the best-performing subset. This approach can achieve high accuracy and captures feature interactions, making it especially useful when feature relationships are essential for prediction. However, it is computationally expensive, prone to overfitting, and requires retraining for different classifiers. Common techniques include the Sequential Selection Algorithms and Heuristic Search Algorithms. The sequential selection algorithms start with an empty set (fullset) and add features (remove features) until the maximum objective function is obtained. [118]To speed up the selection, a criterion is chosen which incrementally increases the objective function until the maximum is reached with the minimum number of features. The heuristic search algorithms evaluate different subsets to optimize the objective function. Different subsets are generated either by searching around in a search space or by generating solutions to the optimization problem.[121, 122]

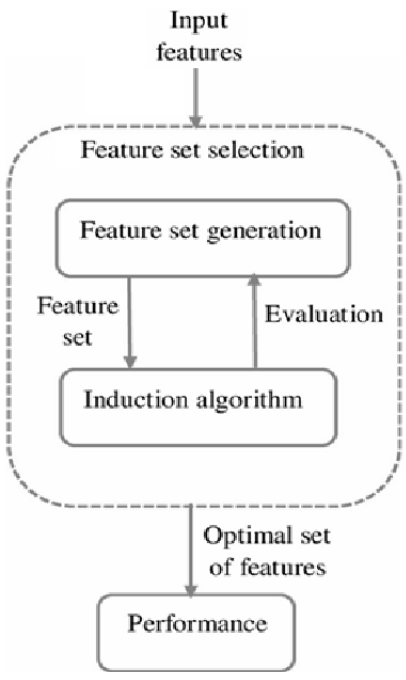


Figure 2.6: Working of Wrapper Technique

Types of Wrapper Method

(a) **Best First (BF)**

Best First is a feature selection method that explores the feature space using a search strategy to identify the subset of features that optimally contribute to model performance. This technique can start from an empty set, a full set, or any random subset of features, allowing flexibility in search direction. It employs a heuristic search strategy, typically a greedy approach, that incrementally adds or removes features based on an evaluation criterion, such as model accuracy or a scoring function that reflects feature relevance.[123].

(b) **Exhaustive Search (ES)**

Exhaustive Search (ES) is a feature selection method that evaluates all possible subsets of features to find the optimal combination for a given model. This approach systematically tests each feature subset, starting from single-feature sets and progressing through all possible combinations up to the full set of features.[124, 125] For each subset, a model is trained and evaluated, usually based on a chosen metric (such as accuracy, F1-score, or cross-validation error). The subset with the best performance on this metric is selected as the final feature set.

How Exhaustive Search Works

- Generate All Subsets: ES begins by generating all possible feature subsets from the dataset. For n features, this results in $2^n - 1$ possible subsets (excluding the empty set), meaning the number of subsets grows exponentially with the number of features.
- Train and Evaluate: For each subset, a model is trained, and its performance is evaluated using the chosen metric. This step is repeated for each subset, ensuring every combination is thoroughly tested.
- Select the Optimal Subset: The subset that yields the highest performance metric is chosen as the optimal feature set for the model.

(c) **Genetic Search (GS)**

Genetic Search (GS) is an advanced feature selection method based on genetic algorithms (GA), inspired by natural selection and evolution principles. It is particularly advantageous for high-dimensional datasets where exhaustive feature selection is computationally infeasible [126]. GS identifies the optimal feature subset by simulating evolutionary processesâselection, crossover, and mutation effectively navigating the feature space without testing every possible combination.

How Genetic Search Works

- Initial Population: The process starts by creating an initial population of feature subsets, represented as binary vectors where each bit denotes the inclusion (1) or exclusion (0) of a feature.
- Fitness Evaluation: Each subset's ”fitness” is evaluated by training a model and assessing its performance on metrics like accuracy or cross-validation error.
- Selection: The highest-performing subsets, or parents, are selected to create the next generation.
- Crossover: Parts of the binary vectors are exchanged between pairs of parents, generating new feature combinations and increasing population diversity.
- Mutation: Mutation flips bits in some subsets to include or exclude random features, further exploring the feature space and reducing the chance of premature convergence.
- Iteration and Stopping Criteria: This process of evaluating, selecting, and breeding new generations is repeated until a stopping criterionâsuch as a fixed number of generations or minimal fitness improvementâis met.
- Optimal Subset Selection: The subset with the best fitness score at the end is selected as the optimal feature set.

(d) **Greedy Stepwise Search (GSS)**

Greedy Stepwise Search (GSS) is a feature selection method that iteratively selects or removes features based on a chosen evaluation metric. GSS is a straightforward approach that operates either in a forward or backward manner to build an optimal feature subset. In forward selection, the algorithm starts with an empty set and adds features one by one; in backward elimination, it starts with all features and removes them one by one. At each step, GSS evaluates the current subset's performance and only adds or removes a feature if it improves the model's performance [127].

How Greedy Stepwise Search Works

- Initial Setup: Depending on the strategy, GSS begins with either an empty set (for forward selection) or a full set (for backward elimination).
- Iterative Addition or Removal:
  - Forward Selection: GSS evaluates each unused feature by adding it to the current subset and assessing the model's performance.
  - Backward Elimination: GSS removes each feature one at a time, observing the effect on model performance.
- Greedy Decision Making: After each evaluation, the feature that most improves the performance is added or removed permanently. If no feature improves the subset, the algorithm stops.
- Stopping Criteria: The process continues until there is no further performance improvement from adding or removing features, or another predefined stopping criterion is met.

(e) **Random Search (RS)**

Random Search for feature selection is a method that explores the feature space by randomly generating and evaluating subsets of features rather than systematically examining every combination. Unlike structured methods like exhaustive or greedy searches, Random Search offers a way to handle high-dimensional feature sets by quickly generating a diverse set of subsets without any particular pattern. It is often used when the feature space is too large to feasibly evaluate each subset, providing a balance between exploration and computational efficiency [127].

How Random Search Works

- Generate Random Subsets: The algorithm starts by generating random subsets of features. Each subset can contain any number of features chosen randomly from the full set.
- Evaluate Subsets: Each randomly generated subset is evaluated based on a specific performance metric, such as accuracy, precision, or cross-validation error. A model is trained with each subset, and the results are recorded.
- Track the Best Subset: The algorithm keeps track of the best-performing subset observed so far. As more subsets are evaluated, the algorithm identifies the subset that yields the highest performance on the chosen metric.
- Stopping Criteria: The process of generating and evaluating random subsets continues until a predefined criterion is met, such as a fixed number of iterations or reaching a performance threshold.

3. **Embedded Methods** Embedded methods are a powerful feature selection approach that integrates selection directly into the model training process as seen in Figure 2.7, making the process efficient and tailored to the model being used. The typical workflow involves training a machine learning model, deriving feature importance from it, and selecting the top predictor variables based on their importance. This approach allows embedded methods to consider feature interactions, similar to wrapper methods, but with faster computation and less risk of overfitting. They also generally yield more accurate results than filter

methods, offering a balanced solution for selecting optimized feature subsets. Popular embedded methods include regularization-based techniques, such as Lasso (L1 regularization) and Elastic Net, which reduce coefficients of less relevant features to zero, which leverage inherent feature ranking capabilities. By combining strengths of both filter and wrapper approaches, embedded methods provide a practical and effective middle ground for feature selection [128].

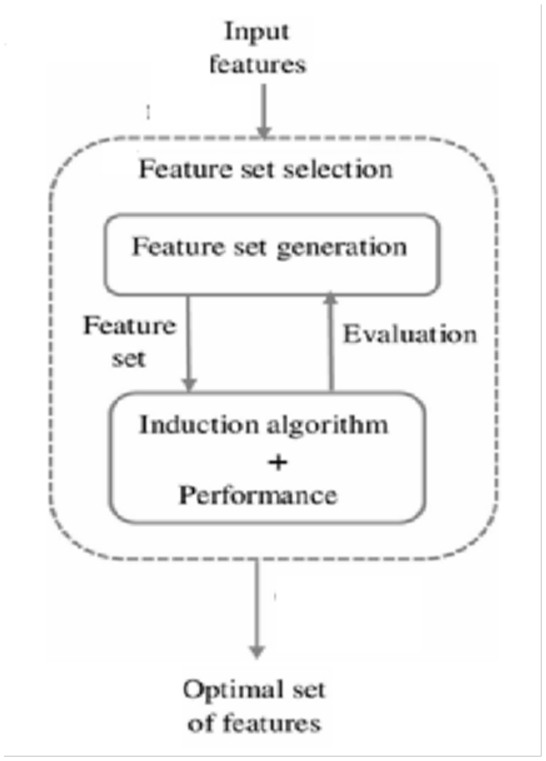


Figure 2.7: Working of Embedded Technique

2.10 Data Balancing

After removing the redundant and irrelevant attributes from data, we observed a significant disparity in the number of data points between the low defect and high defect classes, resulting in imbalanced datasets (see Table 2.4). The non-defect modules contained a substantially greater number of instances compared to the defective modules, indicating a potential bias that could skew the predictions of any model trained on this data. This imbalance is critical to address because it can lead to models that perform well in predicting non-defective instances while failing to accurately identify defective cases, thus undermining the overall effectiveness of Software Defect Prediction (SDP) models. To develop effective and unbiased Software Defect Prediction (SDP) models, it is crucial to tackle the issue of imbalanced data. In this thesis, we employ the Synthetic Minority Over-sampling Technique (SMOTE) to address the imbalance in the dataset. SMOTE is an advanced oversampling method that generates synthetic instances of the minority class, in this case, the defective modules, by interpolating between existing instances. This process involves selecting a sample from the minority class and identifying its nearest neighbours. New synthetic samples are then created by calculating the differences between the sample and its neighbours, adding these differences to the original sample to produce new instances that are similar but not identical [129, 130].

By augmenting the minority class with these synthetic examples, SMOTE helps to balance the dataset, reducing the bias that can lead to skewed predictions. This approach not only increases the number of defective instances available for training but also enhances the model's ability to learn the underlying patterns associated with defects. Consequently, models trained on this balanced dataset are better equipped to accurately predict both defective and non-defective modules, thereby improving their overall predictive performance and reliability in real-world software engineering applications. Through this method, we aim to mitigate the effects of data imbalance and contribute to the development of robust and effective SDP models [100].

2.11 Prediction Model Development and Validation

This thesis develops software defect prediction models in order to identify defect prone modules in future and yet unseen releases of the software. The prediction models learn from historical data of a project to identify which modules are likely to be defective in future releases. The models are developed using supervised machine learning techniques and use various data analysis techniques as discussed in Section 2.6. The training data for the model consists of both the independent variables (software metrics) and the target variable (âdefectiveâ or ânon-defectiveâ). The data analysis technique helps in learning the model. After the model is constructed, it is validated. The validation data is such that only the value of independent variables is provided to the model and the model is supposed to predict a label. The predicted label is matched with the actual label to ascertain the performance of the model. There exist various validation methods such as Leave-one-out, Hold-out, Cross validation and Bootstrapping method [4].

But in this thesis, we utilized the 10-fold cross-validation method because it strikes an effective balance between bias and variance. 10-fold cross-validation provides a comprehensive evaluation by ensuring every data point is used for both training and testing across multiple iterations, reducing the variance in performance estimates through averaging. This approach balances the bias-variance trade-off effectively, offering a stable and reliable estimate of model performance. Additionally, by repeating the process, the reliability of the results is further enhanced, making 10-fold cross-validation a robust choice for evaluating predictive models [131].
Working

In 10-fold cross-validation:

1. The dataset is randomly divided into 10 equal-sized subsets or ”folds”.

2. The model is trained and tested 10 times:
    - In each iteration, 9 folds are used for training and 1 fold is used for testing.
    - Each fold serves as the test set exactly once.
3. Performance metrics (e.g., accuracy) are calculated for each of the 10 iterations.
4. The average performance across all 10 iterations is used as the final estimate.

2.12 Performance Measures

In this thesis, performance metrics play a crucial role in evaluating the effectiveness of software defect prediction (SDP) models. These metrics provide quantitative measures to assess how well a model distinguishes between defect-prone and non-defect-prone modules. By examining the predictions made by each model against the actual data, performance metrics offer insights into various aspects of model quality, such as accuracy, precision, recall, and overall reliability. Selecting appropriate metrics is essential, as they reflect the modelâs ability to balance false positives and false negatives, ensuring that defect predictions are both precise and practical for real-world applications. This section outlines the specific performance metrics utilized in this study and their relevance to achieving robust software defect prediction. In this thesis, several performance metrics are used to evaluate the effectiveness of our SDP models. These metrics include the confusion matrix, accuracy, ROC-AUC (Receiver Operating Characteristic - Area Under the Curve), and MCC (Matthews Correlation Coefficient). Below, each metric is explained in detail along with its mathematical formula.

Several performance metrics are used to evaluate the effectiveness of our SDP models. These metrics include the confusion matrix, accuracy, ROC-AUC (Receiver Operating Characteristic - Area Under the Curve), and MCC (Matthews Correlation

Coefficient). Below, each metric is explained in detail along with its mathematical formula.

1. **confusion matrix**

   The confusion matrix is a powerful tool for evaluating the performance of classification models, providing a detailed summary of correct and incorrect predictions. It organizes the results of a binary classification task into four distinct categories: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Each of these components reveals specific aspects of model behavior and contributes to a deeper understanding of model performance [132, 133].

   Components of the Confusion Matrix

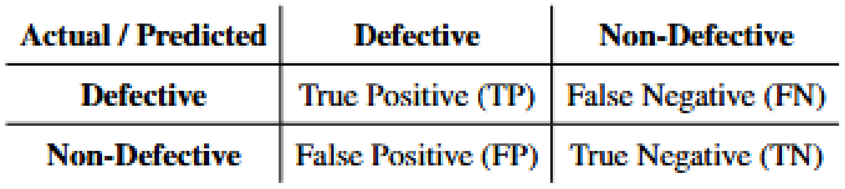
   (a) True Positives (TP): Cases where the model correctly predicts a positive class (e.g., predicting a module as ”defective” when it is indeed defective).

   (b) True Negatives (TN): Cases where the model correctly predicts a negative class (e.g., predicting a module as ”non-defective” when it is indeed non-defective).

   (c) False Positives (FP) (Type I Error): Cases where the model incorrectly predicts the positive class (e.g., predicting a module as ”defective” when it is actually non-defective).

   (d) False Negatives (FN) (Type II Error): Cases where the model incorrectly predicts the negative class (e.g., predicting a module as ”non-defective” when it is actually defective).

   The confusion matrix is typically represented as a 2x2 grid as shown in Figure 2.8.



Actual / Predicted	Defective	Non-Defective
Defective	True Positive (TP)	False Negative (FN)
Non-Defective	False Positive (FP)	True Negative (TN)

Figure 2.8: Structure of Confusion Matrix

2. **Accuracy**

   Accuracy is a commonly used metric that calculates the proportion of correct predictions (both TP and TN) over the total number of predictions:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.39}$$

   While accuracy provides a general measure of model performance, it may not be ideal for imbalanced datasets, as it does not distinguish between types of errors [134].

3. **ROC-AUC (Receiver Operating Characteristic - Area Under the Curve)**

   The ROC-AUC evaluates the model's ability to distinguish between classes. The ROC curve plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The AUC (Area Under the Curve) score provides a single value ranging from 0 to 1, where a score closer to 1 indicates better model performance [135].

$$\text{TPR} = \frac{TP}{TP + FN} \tag{2.40}$$

$$\text{FPR} = \frac{FP}{FP + TN} \tag{2.41}$$

$$\text{AUC} = \int_0^1 \text{TPR}\, d(\text{FPR}) \tag{2.42}$$

A value of 1 represents perfect classification, while 0.5 represents random guessing.

4. **Matthews Correlation Coefficient (MCC)**

   The Matthews Correlation Coefficient is a balanced measure [136] that takes into account all four values in the confusion matrix (TP, TN, FP, and FN):

$$\text{MCC} = \frac{(TP \cdot TN) - (FP \cdot FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{2.43}$$

   The MCC ranges between -1 and +1, where +1 indicates perfect prediction, 0 indicates random prediction, and -1 indicates total disagreement.

Each of these metrics provides unique insights, but together they give a comprehensive view of the model's performance. High accuracy indicates general predictive power, ROC-AUC confirms the model's ability to discriminate between classes, and MCC confirms the model's robustness even with imbalanced data. Therefore, a ”good” SDP model would exhibit high values across these metrics, reflecting reliability and effectiveness in identifying defect-prone modules.

2.13 Statistical Analysis Techniques

In the field of software defect prediction, it is essential to evaluate and validate various hypotheses to ensure the reliability and robustness of the developed models. To achieve this, we employ the Friedman statistical test, followed by the Wilcoxon signed-rank post-hoc test. These non-parametric tests are advantageous because they do not rely on stringent assumptions regarding the underlying data, such as normality or homogeneity of variances, which are often required for parametric tests. Furthermore, as highlighted by Lessmann et al.[137], there is a notable scarcity of studies that rigorously assess the statistical significance of results when comparing different defect prediction models. Therefore, conducting thorough statistical analyses is crucial for enhancing the validity of our conclusions and providing a more solid foundation for the effectiveness of the proposed software defect prediction methodologies.

2.13.1 Friedman Test

Friedman test is used to rank the performance of k techniques over multiple datasets [138]. It is based on the assumption that the performance measures of techniques computed over different datasets are independent of each other. The Friedman test hypothesis can be stated as follows:

- Null Hypothesis (H_0): The performance of different techniques is not statistically different from each other.
- Alternate Hypothesis (H_a): The performance of different techniques is significantly different from each other.

The Friedman test is based on the chi-square statistic (χ^2), which can be computed as follows:

Step 1:For a specific dataset, sort the performance values of all the techniques in descending order. Allocate ranks to each technique based on their performance on the specific dataset. Rank '1' is designated to a technique with the best performance, and rank 'k' is designated to the technique with the worst performance. In case two techniques have equivalent performance on the dataset, assign the average of the ranks that would have been assigned to the techniques.

Step 2:Compute the total of ranks allocated to each technique on all the datasets. The total ranks allocated to each technique are denoted by $R_1, R_2, R_3, \ldots, R_k$.

Step 3:Compute the χ^2 statistic according to the following formula:

$$\chi^2 = \frac{12n}{k(k+1)} \sum_{i=1}^{k} \left(R_i^2\right) - 3n(k+1) \tag{2.44}$$

where R_i is the rank total of the ith technique and n is the total number of datasets. The degree of freedom of the Friedman test is

2.13.2 Wilcoxon Signed Rank Test

This test is used either as a post-hoc test after the results of the Friedman test are found significant or as an independent test to compare the pairwise performance of two techniques. The test is applicable only when two different techniques are evaluated on the same set of datasets [139]. The Wilcoxon test hypothesis can be stated as follows:

- Null Hypothesis (H_0): The performance of the two compared techniques is not statistically different from each other.
- Alternate Hypothesis (H_a): The performance of the two compared techniques is significantly different from each other.

To conduct the test, we first compute the differences among the related pair of values of both techniques. The resulting differences are ranked based on their absolute values. The ranks are allocated according to the following rules:

- Remove the pairs where the difference between both techniques is zero. The number of reduced pairs is denoted as n_r.
- Assign a rank of â1â to the smallest absolute difference and so on to all the n_r pairs.

- In case of a tie, an average of tied ranks is allocated.

Next, two variables R^+ and R^- are computed:

- R^+ is computed as the sum of ranks where the difference is positive (i.e., the first technique outperforms the second technique).
- R^- is computed as the sum of ranks where the difference is negative (i.e., the second technique outperforms the first technique).

The Z statistic is then computed as follows:

$$Z = \frac{Q - \mu}{\sigma} \tag{2.45}$$

where $Q = \min(R^+, R^-)$, $\mu = \frac{n_r(n_r+1)}{4}$, and $\sigma = \sqrt{\frac{n_r(n_r+1)(2n_r+1)}{24}}$.

If the Z statistic is in the critical region at a specific level of significance (e.g., $\alpha = 0.05$), we reject the null hypothesis. This means that the performance of the two compared techniques is significantly different from each other.

The Wilcoxon test was performed with Bonferroni correction to account for family-wise error. With Bonferroni correction, a p-value is considered significant only if it is less than α/c, where c is the total number of comparisons performed.

To evaluate the practical significance of the obtained results, we also report the effect size of the Wilcoxon test for significant cases. The effect size was computed using the following formula:

$$r = \frac{Z}{\sqrt{N}} \tag{2.46}$$

where N is the total number of observations.

As indicated by Cohen [140], an effect size value of 0.1 is considered small, 0.3 is considered medium, and 0.5 is considered large.

Chapter 3

A Systematic Review of Feature Reduction Techniques for Software Quality Predictive Modelling using Object-Oriented Metrics

3.1 Introduction

Quality of software can be measured in terms of many attributes such as reliability, maintainability, defect proneness, effort, and change proneness. To predict these attributes in the upcoming version of the software, Software Quality Predictive Models (SQPM) are constructed with the help of software metrics and quality attributes. There are plenty of object-oriented metrics proposed by researchers in the past to measure these quality aspects, such as QMOOD's metrics, Chidamber & Kemerer metrics, Lorenz & Kidd metrics, Li & Henry, and Briand et al., to name a few. These SQPMs quantitatively describe how the internal structural properties relate to relevant external software quality attributes such as maintainability [141].

However, according to researchers in [142, 143], there are some grave issues that need to be fixed before using the software quality prediction results to escort the quality assurance process. One main concern is about identifying a subgroup of software attributes that are considerably correlated with software quality attributes. Many studies in Software Quality Prediction have stated a remarkably higher misprediction rate that might be credited to some in-built issues linked with the project corpus. One of the issues may be that the dataset contains unnecessary and redundant information, and therefore, there is a requirement for preprocessing before using them for quality prediction.

Many studies show that a large number of features may result in decreased accuracy and high errors as compared to a reduced set of features. The high dimensionality of the dataset can also result in high computational cost and memory usage. As the efficiency of the Software Quality Prediction model is dependent on a group of software metrics used for the model building process, the choice of the correct set of software features is an essential and primary task[144].

Feature selection is a process of choosing a reduced set of relevant features from available features such that the efficiency of the prediction model can be improved. Each feature selection method analyses the available features and adopts a relevant set of features. The criteria used to search the useful features depend on the nature of the technique used. The author incorporated a software metric reduction technique, such as Principal Component Analysis (PCA), to advance the efficiency of the models and to decrease multicollinearity. In [145], Malhotra et al. used Correlation-based Feature Selection (CFS), which helps to determine redundant and undesirable noisy features.

To identify the association between the feature reduction techniques and Software Quality predictive modelling, it is essential to systematically examine empirical studies reported in the literature about the use of feature reduction techniques for the development of Software Quality Predictive Modelling. In order to do so, we perform a systematic review of existing studies in this area. We investigated the following Research Questions (RQs), and the motivation behind them is provided:

Table 3.1: Research Questions and Motivations

RQ #	**Research Questions**	**Motivation**
RQ1	Which feature reduction techniques have been widely used for SQPM?	Identifies the commonly adopted feature reduction techniques by researchers.
RQ2	Which Object-oriented metrics are mostly used in studies based on feature reduction in SQPM?	Identifies the commonly used object-oriented metrics by researchers in the field of feature reduction for SQPM.
RQ3	Which Object-oriented datasets are commonly used in studies based on feature reduction in SQPM?	Identifies the popularly used datasets by researchers in the field of feature reduction for SQPM.
RQ4	Which feature reduction techniques in combination with machine learning techniques have been widely used for software quality attribute prediction?	Explores the compatibility of feature reduction techniques with machine learning techniques.
RQ5	Which performance metrics are used by experiments for analyzing the developed SQPM based on feature reduction?	Identifies the performance metrics used in literature for evaluating a specific software quality predictive model.
RQ6	Which validation method has been used for SQPM based on feature reduction?	Identifies the validation methods commonly employed in the literature to assess the effectiveness of software quality prediction models (SQPM) that incorporate feature reduction techniques.

The review also helps in the identification of research gaps and provides future guidelines to researchers and practitioners. The aim of the review is to systematically summarize the empirical evidence reported in the literature with respect to various metrics, datasets, data analysis techniques, performance measures, validation methods, and statistical tests used for software change prediction.

The chapter is organized as follows: Section 3.2 describes the review procedure and the stages involved in conducting the review. Section 3.3 states the review protocol. Section 3.4 provides the answers to each of the investigated RQs. Finally, Section 3.5 discusses the future directions.

3.2 Review Procedure

According to the guidelines advocated by Kitchenham et al. [146], a review is conducted in three fundamental stages. These stages are reportedly planning, conducting and reporting. The foremost step of the planning stage is to evaluate the necessity of the review. As already discussed, the aim of the review was to assess and summarize the empirical evidence in the domain of software change prediction. It intends to provide an overview of existing literature in the domain and would scrutinize possible future directions. Once the need of the review is assessed, the planning stage involves formation of RQs. Thereafter, a review protocol is formulated. The protocol includes a detailed search strategy. The search strategy consists of the list of possible search databases one intends to scrutinize, the search string and the criteria for including and excluding the extracted studies. Apart from the search strategy, the protocol also includes the criteria for assessing the quality of the candidate studies, the procedure for collecting the relevant data from the primary studies and synthesis of the collected data. The second stage involves the actual execution of the review protocol. In this stage, all the relevant literature studies are extracted, scrutinized and the relevant data is obtained. The final stage of the review reports the results of the investigated RQs.

The RQs are answered on the basis of the data extracted from primary studies.

3.3 Review Protocol

The review protocol includes the search strategy, inclusion and exclusion criteria, and the quality criteria for assessing the collected candidate studies. The following sections describe the review protocol followed.

3.3.1 Search Strategy

In order to design our search terms, we divided the explored RQs into comprehensive logical units. Moreover, terms were identified from paper titles, keywords, and abstracts. Thereafter, all equivalent terms and synonyms were compiled using Boolean OR, while distinguishable search terms were aggregated using Boolean AND. The period of the search was chosen from January 2000 to December 2020. The search string for extracting candidate studies is as follows:

"Software" OR "Application" OR "Project" OR "Open-source project" OR "Product" OR "Object-oriented" OR "empirical" OR "quality" OR "coupling" OR "cohesion" OR "class" OR "inheritance" AND ("Bug" OR "Defect" OR "Fault" OR "Error" OR "Cost" OR "Maintainability" OR "Maintenance" OR "Change" OR "Size") AND ("proneness" OR "prone" OR "prediction" OR "probability" OR "classification" OR "estimation" OR "investigation") AND ("Features" OR "Metrics" OR "Attribute" OR "Variable") AND ("Reduction" OR "Selection" OR "Extraction" OR "Subset").

We searched a number of prominent search databases such as SCOPUS, Wiley, SpringerLink, IEEExplore, ScienceDirect, and ACM Digital Library. We also searched the reference lists of the extracted studies. As a result of this comprehensive effort, we identified 115 relevant studies. These studies were then subjected to the inclusion and exclusion criteria, as described in Section 3.3.2.

3.3.2 Inclusion and Exclusion Criteria

The following inclusion and exclusion criteria were applied to the identified studies:

Inclusion Criteria

- Empirical studies using feature selection or extraction techniques.
- Empirical studies using datasets based on Object-oriented metrics.
- Empirical studies comparing the performance of feature selection or extraction techniques, machine learning techniques, deep learning, and statistical techniques.
- Empirical studies proposing hybrid techniques by combining machine learning techniques with some evolutionary feature reduction techniques.

Exclusion Criteria

- Studies on software quality prediction without empirical analysis of results.
- Empirical studies without any dataset based on Object-oriented metrics.
- Empirical studies where the dependent variable was other than fault/bug/defect, change, or maintainability.
- Review studies.
- In case a conference paper was extended in a journal, only the journal version of the paper was included.

The inclusion and exclusion criteria mentioned above were analyzed by two different researchers to arrive at the same decision after meetings and discussions. In cases of differing views, in-depth scrutiny of the complete text was performed to reach an agreement on inclusion or exclusion for that article. By implementing the above-mentioned inclusion and exclusion conditions, 26 studies were selected.

3.3.3 Quality Criteria

It is important to analyse the significance and contribution of each selected study in answering the various RQs. To attain assurance in the applicable studies, a quality questionnaire is developed which contains 15 quality-based questions as given in Table to measure their impact and offer weightage to the studies with respects to the research questions of the systematic review. The Table includes the complete framed 15 questions and the percentage of primary studies answering these quality assessment questions. The studies are graded between 0 to 1. 1 implies Yes (Y), 0 implies NO (N) and 0.5 denotes Partly (P). For each study, the grade for each question is aggregated to compute total marks. At max, any study could score a maximum of 15 and a minimum of 0. All the studies whose QS was less than 7.5 (50% of the total quality score) were rejected. After this step, a total of 22 literature studies were selected, which were termed as primary studies of our review. Table 3.2 lists all the primary studies with a specific allocated study number and its QS. Relevant data pertaining to RQs was extracted from these studies and the obtained results are reported in Section 3.4

3.4 Results Analysis

This section states the results of the review. Around 56% of publications were from journals, and 44% were from reputed national and international conferences. Table 3.3

Table 3.2: List of Quality Assessment Questions

SN.	**Quality Assessment Question**
Q1	Whether the objective of the study is clear?
Q2	Does the study add value to the existing literature?
Q3	Is the dataset size sufficient for this type of study?
Q4	Does the study perform the pre-processing analysis?
Q5	Are the independent and dependent variables clearly defined?
Q6	Is the data collection procedure clearly defined?
Q7	Does the study use statistical tests for result evaluation?
Q8	Are the experiment details clearly defined?
Q9	Are the validity threats given?
Q10	Does the study provide parameters of the test?
Q11	Are the drawbacks of the study given?
Q12	Does the study clearly define the performance parameters used?
Q13	Are the learning techniques clearly defined?
Q14	Are the results clearly stated?
Q15	Does the abstract provide sufficient information about the content of the study?

lists the most popular journal and conference venues.

The quality assessment questions were allotted scores that were distributed into three groups:

- High: $13 \leq$ scores ≤ 15
- Medium: $10 \leq$ scores ≤ 12
- Low: $0 \leq$ scores ≤ 9

The Table 3.4 below represents a unique identifier for every chosen primary study. These exclusive identifiers will be used in all successive segments to refer to their equivalent selected primary study. A total of six primary studies, i.e., PS12, PS16, PS18, PS19, PS20, and PS21, attained the highest scores. Keen readers can go through these studies for further reading. Studies with low scores were discarded, and as a result, four studies with a score of ≤ 7.5 were removed from the relevant studies, resulting in a total of 22 primary studies deemed suitable for the systematic review.

3.4.1 Result Analysis based on RQ1

RQ1: Which Feature Reduction Techniques Have Been Widely Used for SQPM?

Feature selection is a method of choosing a subset of significant features or attributes from the available features of the dataset. Feature selection techniques can be grouped into various categories such as wrappers, filters, and embedded techniques.

Wrappers refer to algorithms that use feedback from a learning algorithm to determine which attribute(s) to use in building a classification model. In contrast, filters select attributes using a method independent of an induction algorithm to identify the most relevant attributes. Embedded techniques integrate the feature selection algorithm as part of the learning algorithm, with the most typical example being the random forest.

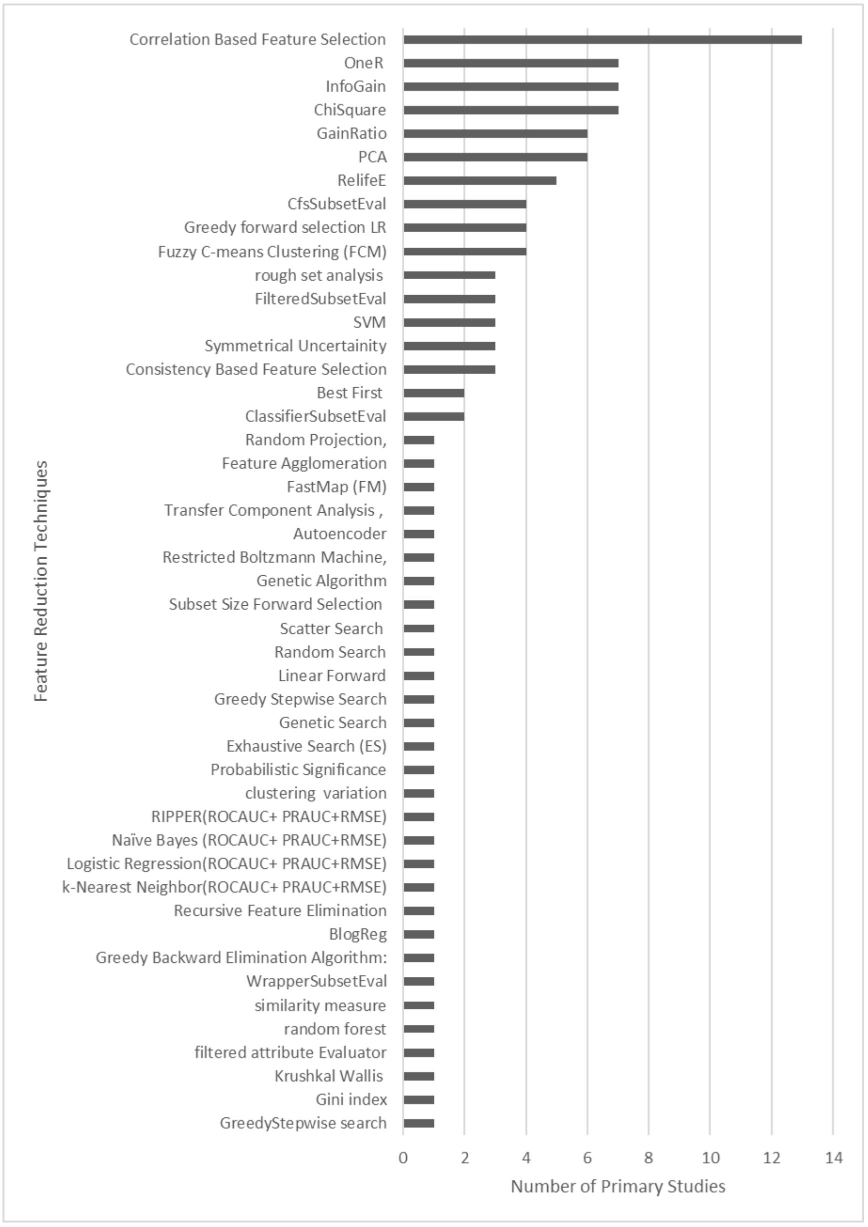


Figure 3.1: Commonly Used Feature Reduction Techniques

Figure 3.1 illustrates the number of studies based on specific categories of feature reduction techniques for SQPM, including change prediction, defect prediction, and maintainability prediction. A total of 47 feature reduction methods have been identified in the selected studies, out of which 17 have been used by two or more primary studies, and the remaining 30 have been used by a single primary study.

The most commonly used feature reduction technique for SQPM is Correlation-Based Feature Selection (59%), followed by InfoGain, ChiSquare, and OneR (32%). Other prominent techniques include PCA and Gain Ratio (27%).

3.4.2 Result Analysis based on RQ2

RQ2: Which Object-Oriented Metrics Are Mostly Used in Studies Based on Feature Reduction in SQPM?

Among the selected primary studies, we identified 47 object-oriented independent metrics (Figure 3.2). Number of Children (NOC) and Depth of Inheritance Tree (DIT) are the most highly studied object-oriented metrics, being used by all primary studies. Following these, Weighted Method per Class (WMC), Lack of Cohesion of Methods (LCOM), and Response For Class (RFC) are the Chidamber and Kemerer (C&K) metrics used in 95%, 95%, and 90% of primary studies, respectively.

On the other hand, coupling metrics such as Ancestor Class MethodâMethod Import Coupling (AMMIC), Descendant Class MethodâMethod Export Coupling (DMMEC), Other Class MethodâMethod Export Coupling (OMMEC), and Other Class MethodâMethod Import Coupling (OMMIC) are evaluated by the least number of primary studies, as they are used in only 4.5% of the studies.

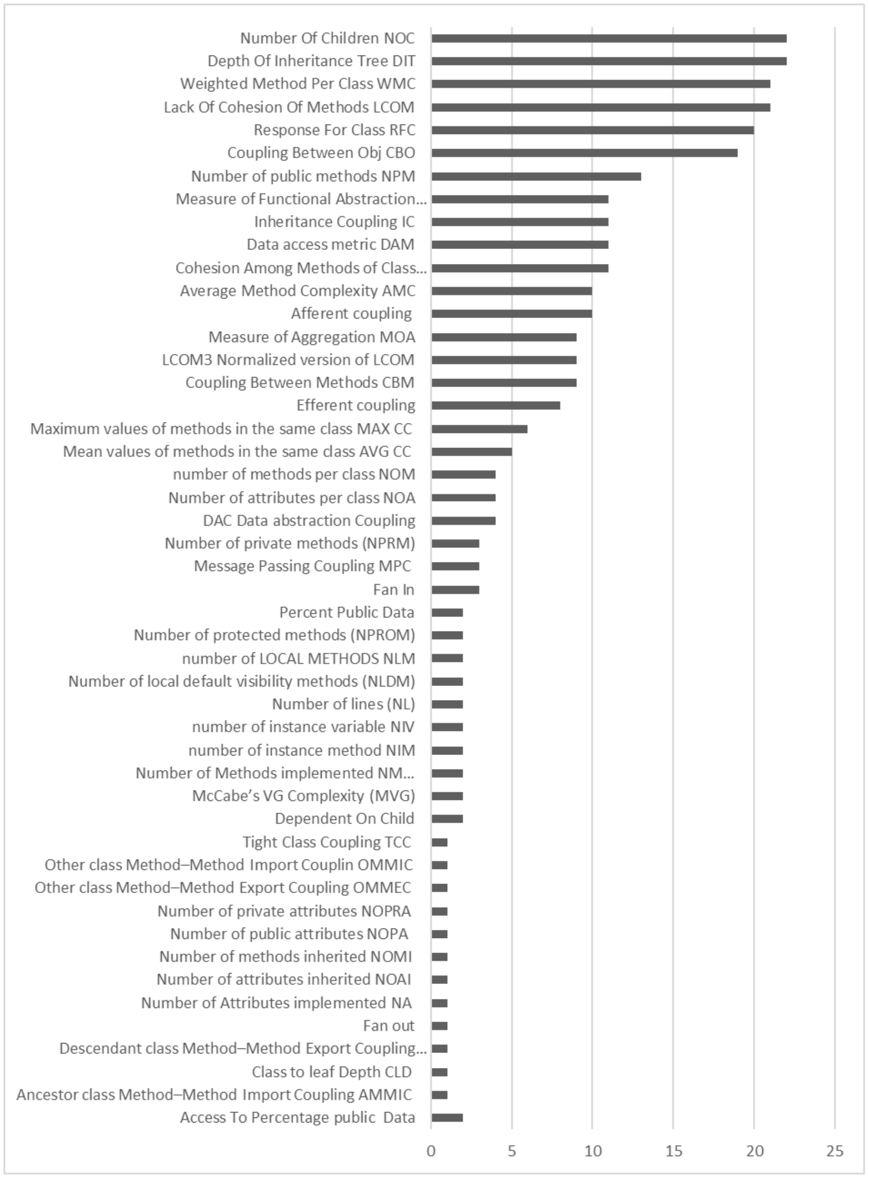


Figure 3.2: Object-Oriented Metrics Used in SQPM Studies

3.4.3 Result Analysis based on RQ3

RQ3:Which Object-oriented datasets are commonly used in studies based on feature reduction in SQPM?

With respect to the software systems and datasets used in the primary studies, it was found (Figure 3.3) that a total of eight unique types of datasets were identified. Among these:

- 11% are open-source projects,
- 8% are student projects,
- 11% are web-based datasets,
- 7% are Eclipse datasets,
- 22% are NASA datasets, and
- 33% are PROMISE datasets.

Key Datasets and Their Types

1. **NASA Datasets:** These datasets are publicly available in the NASA repository provided by the NASA Metrics Data Program. They are the most used datasets for SQPM and include projects such as KC1, KC2, and PC1, among others.
2. **Student Data:** These include academic software developed by students from different universities.
3. **PROMISE:** These datasets are freely available in the PROMISE repository (`https://code.google.com/p/promisedata/`) and include software projects such as Camel, Ivy, and Xerces.
4. **Open-Source Projects:** This category involves studies that use open-source projects such as Frinika, FreeMind, Drumbox, and Drumkit, among others.
5. **Commercial Datasets:** This category includes industrial and private datasets, such as those from a commercial bank, a commercial Java application, and

telecommunication companies. Examples include the User Interface System (UIMS) and Quality Evaluation System (QUES).

6. **Eclipse:** The Eclipse dataset includes Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, and Mylyn projects. These are part of an open-source project whose defect datasets are publicly available. Eclipse is an integrated development environment developed in Java.

7. **Android:** Android packages such as Contacts and Gallery are used for the prediction of software attributes.

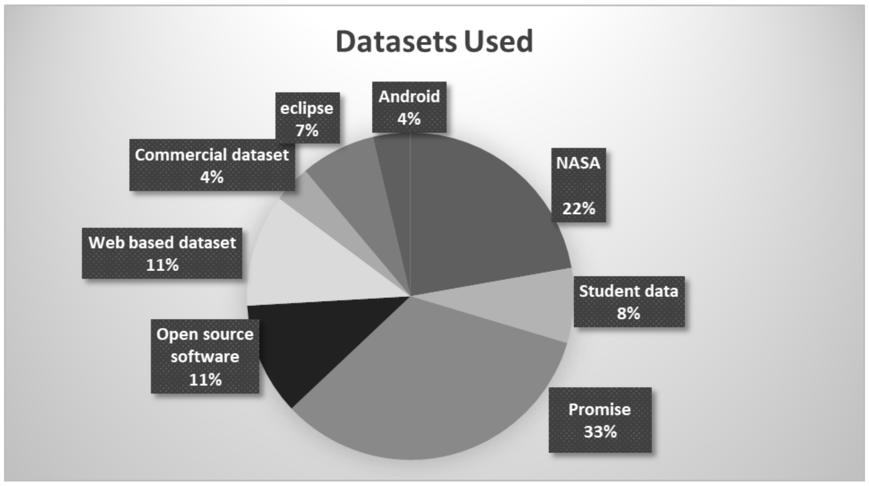


Figure 3.3: Types of Datasets Used in Primary Studies

3.4.4 Result Analysis based on RQ4

RQ4: Which feature reduction techniques in combination with machine learning technique have been widely used for Software quality attribute Prediction?

According to Figure 3.4, a total of 46 unique machine learning techniques were identified. These techniques belong to various categories such as neural networks, ensemble-based techniques, and evolutionary techniques.

Approximately 45% of primary studies utilize Random Forests, Logistic Regression, and NaÃ¯ve Bayes, followed by Decision Trees, which are used by around 32% of primary studies. Other notable techniques include Support Vector Machines and ensemble techniques such as boosting.

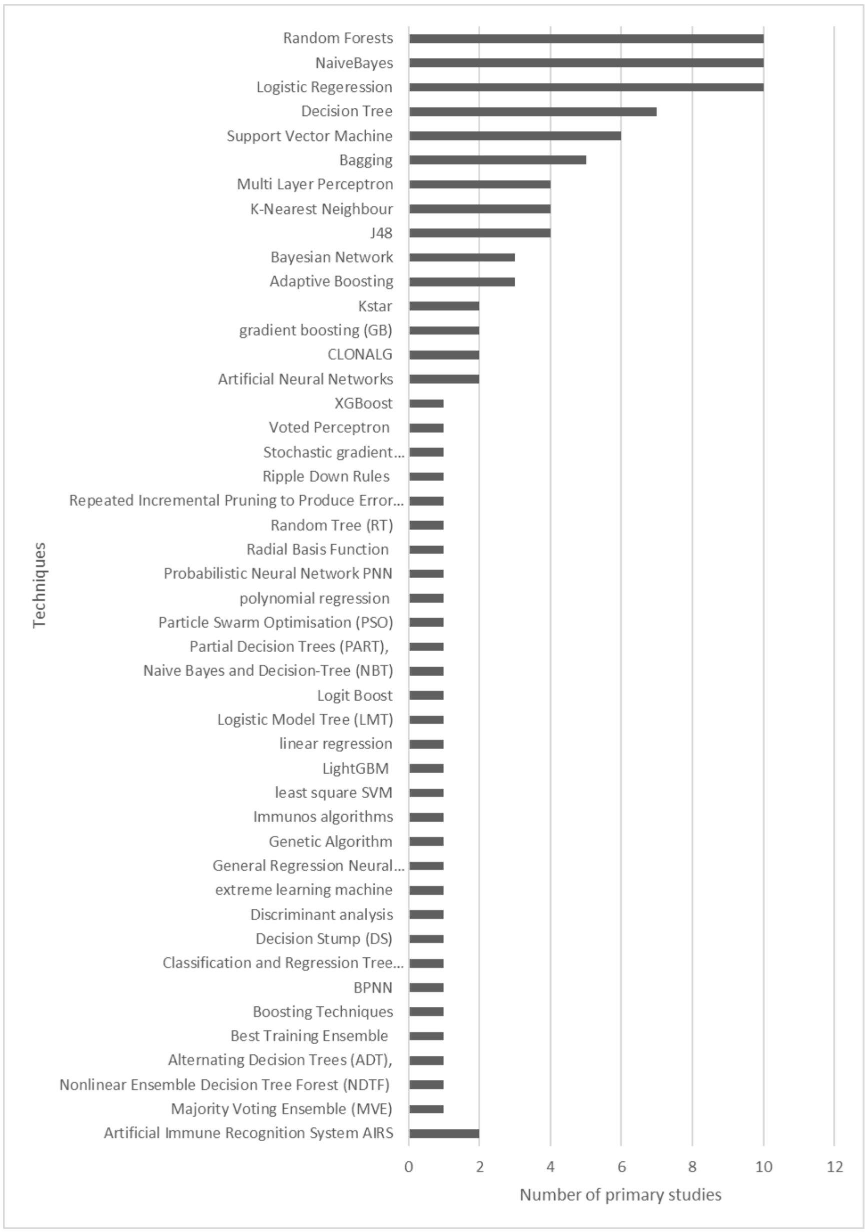


Figure 3.4: Machine Learning Techniques Used in Primary Studies

3.4.5 Result Analysis based on RQ5

RQ5: Which performance metrics are used by experiments for analysing the developed SQPM based on feature reduction? In order to develop a predictive model and evaluate its effectiveness, it is essential to use an efficient performance measure.

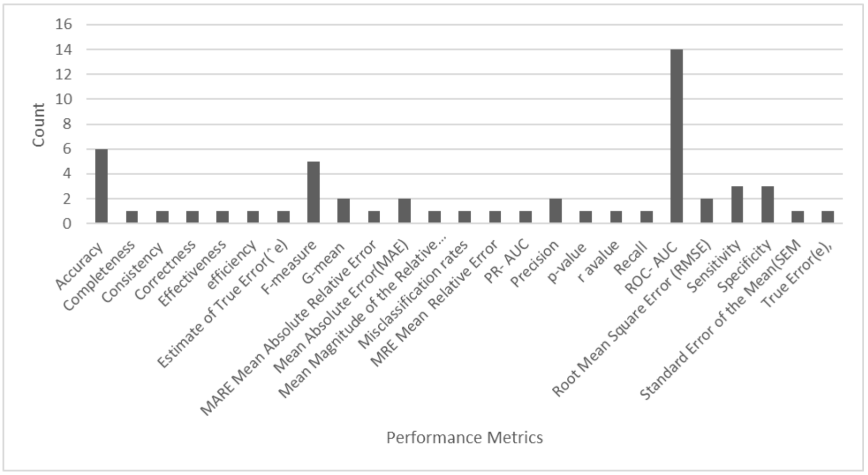


Figure 3.5: Performance Measures Used in Primary Studies

Figure 3.5 illustrates the performance measures identified in the selected primary studies. Among these, we identified 25 unique performance metrics. Some of these measures are used for regression-based problems, such as maintainability, while others are used for binary classification problems, such as the prediction of change proneness and defect proneness.

Nearly 63% of primary studies use the ROC-AUC measure for performance evaluation. The second most popular measure is accuracy (27%), followed by F-measure (23%).

3.5 Discussion

In this study, we performed a systematic literature review to analyze and assess the performance of feature reduction techniques for Software Quality Predictive Modelling (SQPM). Following a systematic series of steps and evaluating the quality of the studies, we identified 22 primary studies (2000â2020). We then summarized the insights from the primary studies based on feature reduction techniques, metrics, machine learning techniques, datasets, and performance measures.

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

- The feature reduction techniques were broadly grouped into filter, wrapper, and embedded categories. The most frequently used feature reduction techniques for SQPM were Correlation-Based Feature Selection, OneR, and InfoGain. Other prominent techniques included PCA and Gain Ratio.

- Two object-oriented metrics, i.e., Number of Children (NOC) and Depth of Inheritance Tree (DIT), were used in all primary studies. Three metrics from the CK metric suite were used in nearly 90% of the studies, whereas coupling metrics like AMMIC, DMMEC, OMMEC, and OMMIC were evaluated in only 4.5% of primary studies.

- The PROMISE dataset was the most frequently used object-oriented dataset in the primary studies. However, it was observed that not many studies utilized commercial or Android datasets for assessing the usefulness of feature reduction methods.

- Various machine learning techniques were identified in the literature for SQPM. About 45% of primary studies used Random Forests, Logistic Regression, and NaÃ¯ve Bayes, followed by Decision Trees, which were used in approximately

32% of primary studies. Support Vector Machines and ensemble techniques, such as boosting, were also prominently used with feature reduction techniques.

- Nearly 63% of primary studies used the ROC-AUC measure for performance evaluation. Accuracy (27%) and F-measure (23%) were the next most popular performance measures.

The following are guidelines for researchers and software practitioners for carrying out future research on Software Quality Predictive Modelling using feature reduction techniques:

1. To achieve more generalizable results, more studies should focus on SQPM using feature reduction techniques. Comparative studies assessing the performance of feature reduction techniques alongside machine learning techniques should be conducted.

2. There is a lack of studies in the literature that examine the effect of feature extraction techniques on software quality attributes using object-oriented metrics. Future research should focus on assessing the prediction efficiency of these seldom-used techniques.

3. More commercial object-oriented datasets should be made publicly available to enable more experiments on freely accessible datasets.

4. Before applying feature reduction techniques to a dataset, researchers should thoroughly understand the decision to use a specific feature reduction technique, the characteristics of the machine learning technique, and the properties of the dataset.

Table 3.3: List of Publication Sources

SN.	Publication Name	Publication Type	Count	Percentage (%)
1	International Conference on Product Focused Software Process Improvement	C	1	4.348
2	Information and Software Technology	J	1	4.348
3	Information Sciences	J	1	4.348
4	International Conference on Multimedia and Information Technology	C	1	4.348
5	Software Engineering: An International Journal	J	2	8.696
6	Int. J. Mach. Learn. & Cyber.	J	1	4.348
7	IEEE International Conference on Recent Advances and Innovations in Engineering	C	1	4.348
8	Information and Software Technology	J	2	8.696
9	Service-oriented Computing and Applications	J	1	4.348
10	IEEE International Conference on Computational Systems and Information Technology for Sustainable Solutions	C	1	4.348
11	IEEE Annual Computer Software and Applications Conference	C	1	4.348
12	Journal of Intelligent & Fuzzy Systems	J	2	8.696
13	IEEE ACCESS	J	1	4.348
14	Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)	C	3	13.043
15	IEEE/ACM International Conference on Mining Software Repositories	C	1	4.348
16	Empirical Software Engineering	J	1	4.348
17	The Journal of Systems and Software	J	1	4.348
18	International Conference on Enterprise Information Systems	C	1	4.348

Table 3.4: List of Selected Primary Studies

Primary Study Identifier	**Paper**
PS 1	Catal (2007)
PS 2	Kanmani (2007)
PS 3	Catal (2009)
PS 4	Jin (2010)
PS 5	Malhotra (2011)
PS 6	Malhotra (2012)
PS 7	Malhotra (2013)
PS 8	He (2014)
PS 9	Singh (2014)
PS 10	Laradji (2015)
PS 11	Muthukumaran (2015)
PS 12	Kumar (2016)
PS 13	Kumar (2016)
PS 14	Nanda (2017)
PS 15	Chen (2017)
PS 16	Ghotra (2017)
PS 17	Malhotra (2018)
PS 18	Malhotra (2019)
PS 19	YU (2019)
PS 20	Kumar (2019)
PS 21	Kondo (2019)
PS 22	Sousa (2019)

Chapter 4

Software Defect Prediction using Feature Extraction Techniques

4.1 Introduction

Software defect prediction (SDP) aims to classify software modules as defective or non-defective, allowing software quality assurance teams to focus resources on the most defect-prone components. Each software module is typically described by a set of metrics or features, but as datasets have grown, many of these features have become redundant or irrelevant. High-dimensional datasets lead to increased computational costs and reduced model performance, a problem known as the ”curse of dimensionality.” According to Bellman, the amount of data required for reliable performance rises exponentially with the number of features, making large feature sets inefficient and difficult to manage [147–149].

Feature extraction [150] is an essential technique to address high-dimensionality in SDP. By transforming the original feature set into a smaller, more relevant subset, feature extraction reduces complexity, removes noise, and improves model interpretability. This technique allows models to retain the most informative features, enhancing computational efficiency and prediction accuracy. Unlike feature selection, which simply removes features, feature extraction generates new, non-redundant features through transformations that preserve essential information. Common feature extraction techniques include Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA), and Kernel-based PCA, which help create a lower-dimensional representation of the data while minimizing information loss.

In this chapter, both linear and non-linear feature extraction methods are investigated to optimize defect prediction models. Reducing the dimensionality of feature space without compromising data quality is crucial for effective defect prediction, as it enables more efficient processing and yields more reliable, interpretable predictions, ultimately supporting better software quality assurance practices. Furthermore, the chapter examines the effectiveness of four feature extraction technique (FE) for software defect prediction model. The results were validated with statistical test.

The structure of this chapter is as follows: Section 4.2 presents the research background and outlines the research methodology. Section 4.3 details the experimental design. Section 4.4 provides the analysis of results, and finally, Section 4.5 discusses the key findings of the chapter. The results of the chapter are published in [151]

4.2 Research Background and Methodology

This section provides insight into the procedures and methods used in this study. Here we have briefly discussed each method required to perform the empirical study such as the process involved in data collection, pre-processing, classification, model validation, measuring performance and selection of statistical test.

4.2.1 Dataset Collection

In this study, nine open-source projects namely Ant - Version 1.7, ArcPlatform - Version 1, Camel - Version 1.6, jEdit - Version 4.3, Log4j - Version 1.2, Poi - Version 3.0, Prop - Version 6.0, Tomcat - Version 6 and Xalan - Version 2.7 from Apache software Foundation Systems which are publicly available in the open-source dataset repository called PROMISE Repository are used as datasets. This is the most used repository in Software Fault Prediction. For this study, different projects of varied size as well as with different defect rate is considered, and these projects have 20 Object-Oriented metrics. Also, there is a defect label for each class which shows whether a module is defective or not. The detailed information in present in Section 2.7.1.

4.2.2 Data Normalisation

Selection of normalization technique for specific data totally depends upon the user. For this comparative study, 20 input variables are not in the same range of magnitude. Hence, we perform data normalization on these attributes using min-max normalization (Detailed in Section 2.8.1) to transform data within the range of [0, 1].

4.2.3 Feature Extraction Techniques

Dimensionality reduction techniques, such as Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA), Kernel-based Principal Component Analysis (K-PCA), and Autoencoders, are applied to reduce the dataset's dimensionality. PCA maximizes variance retention while reducing dimensions, improving computational efficiency. LDA focuses on maximizing class separation, making it suitable for distinguishing defective from non-defective instances. K-PCA captures non-linear relationships, allowing for flexibility in modelling complex data patterns. Autoencoders utilize deep learning to learn compressed representations, effectively uncovering intricate feature interactions. This combination of techniques addresses high dimensionality challenges and enhances model performance and interpretability in defect prediction. These techniques are detailed in Section 2.9.1.

4.2.4 classification Technique

The choice to focus on Support Vector Machine (SVM) in this study stems from its proven effectiveness in handling classification tasks, particularly in the context of software defect prediction. SVM is known for its ability to create robust decision boundaries that maximize the margin between classes, making it particularly suitable for datasets with clear class separations. Details of this method is given in subsection 3.8. Additionally, SVM performs well with high-dimensional data, which is common in software metrics [152]

4.2.5 Model validation Technique

For model validation, we employed the 10-fold cross-validation method. The detailed working can be found in Section 2.11.This approach was chosen to ensure robust evaluation of model performance by dividing the dataset into ten subsets, allowing each subset to serve as a test set while the remaining nine are used for training. This technique minimizes overfitting and enhances generalization.

4.2.6 Performance Measures

Accuracy provides a straightforward assessment of the model's overall correctness, while ROC-AUC evaluates the trade-off between sensitivity and specificity, offering a comprehensive understanding of the model's discriminative ability across different thresholds. Hence these two methods are employed and comprehensive description can be found in Section 2.12

4.2.7 Statistical Test

To statistically evaluate the performance of the feature extraction technique, we employ the Friedman test (A full description can be found in Section 2.13). This non-parametric test is advantageous due to its less stringent assumptions, allowing for the potential oversight of anomalies within the datasets, as well as issues related to homogeneity of variance and normality of distributions.

4.3 Experimental Design

This section outlines the process of variable selection, distinguishing between independent and dependent variables essential for the study. It also details the formulation of hypotheses that guide the research direction. Additionally, the tools and methodologies employed for conducting experiments are discussed, ensuring a robust framework for data analysis.

4.3.1 Variable Selection

The independent variables used in this study consist of 20 software metrics, including Lines of Code (LOC) and CK metrics. Examples of these metrics are Response for the Class (RFC), Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Lack of Cohesion in Methods (LCOM), Afferent Coupling (CA), Efferent Coupling (CE), and Number of Public Methods (NPM). A comprehensive overview of these independent variables is provided in Section 2.5. The dependent variable in this study is defect_proneness, which indicates whether a class is defect-prone or not.

4.3.2 Hypothesis Formulation

The following set of hypotheses has been developed to evaluate the defect prediction model using machine learning techniques based on feature extraction.

- *Null Hypothesis (H0):* The ROC-AUC results of the defect prediction models developed using SVM does not show any significant difference when no feature extraction method or four different feature extraction methods (PCA, LDA, K-PCA, Autoencoders) are used for the given datasets.
- *Alternate Hypothesis (H1):* The ROC-AUC results of the defect prediction models developed using SVM shows the significant difference when no feature extraction method or four different feature extraction methods (PCA, LDA, K-PCA, Autoencoders) are used for the given datasets.

4.4 Results Analysis

4.4.1 Result Analysis based on RQ1

RQ1:How do different feature extraction techniques (PCA, LDA, K-PCA, Autoencoders) impact the accuracy of Support Vector Machine (SVM) models in defect prediction across various software projects?

According to the value of accuracy performance measures in Table 4.1 and Figure 4.1, the analysis of the results for the various feature extraction techniques combined with Support Vector Machine (SVM) classification reveals distinct performance trends across different projects.

The results indicate that the combination of PCA with SVM generally yields competitive accuracy across most projects, achieving high scores particularly in the Jedit (0.975) and Xalan (0.985) datasets. Linear Discriminant Analysis (LDA) combined with SVM consistently provides solid results, particularly excelling in projects such as Ant (0.828) and Arc (0.897). However, it slightly underperforms compared to PCA in certain cases, such as Camel (0.801) and Log4j (0.902).

While Kernel PCA (K-PCA) shows good results, it appears less effective than PCA in projects like Jedit (0.877) and Poi (0.632). Its performance might suggest challenges in capturing complex patterns in some datasets. Using only SVM without dimensionality reduction techniques yielded varying results. For instance, it performed best in the Arc (0.898) and Jedit (0.970) datasets but was outperformed by other combinations in the Camel (0.796) and Poi (0.765) datasets. Among the feature extraction methods, Autoencoders (AE) combined with SVM also show strong performance, especially with Jedit (0.977) and Prop (0.903).

Table 4.1: Accuracy Calculated for Each Project

Projects	**PCA**	**LDA**	**K-PCA**	**AE**	**Without FE**
Ant	0.808	0.828	0.776	0.776	0.827
Arc	0.880	0.897	0.880	0.881	0.898
Camel	0.800	0.801	0.801	0.811	0.796
Jedit	0.975	0.957	0.877	0.977	0.970
Log4j	0.925	0.902	0.892	0.922	0.922
Prop	0.853	0.902	0.893	0.903	0.902
Poi	0.765	0.774	0.632	0.706	0.765
Tomcat	0.907	0.908	0.906	0.909	0.906
Xalan	0.985	0.986	0.878	0.987	0.978

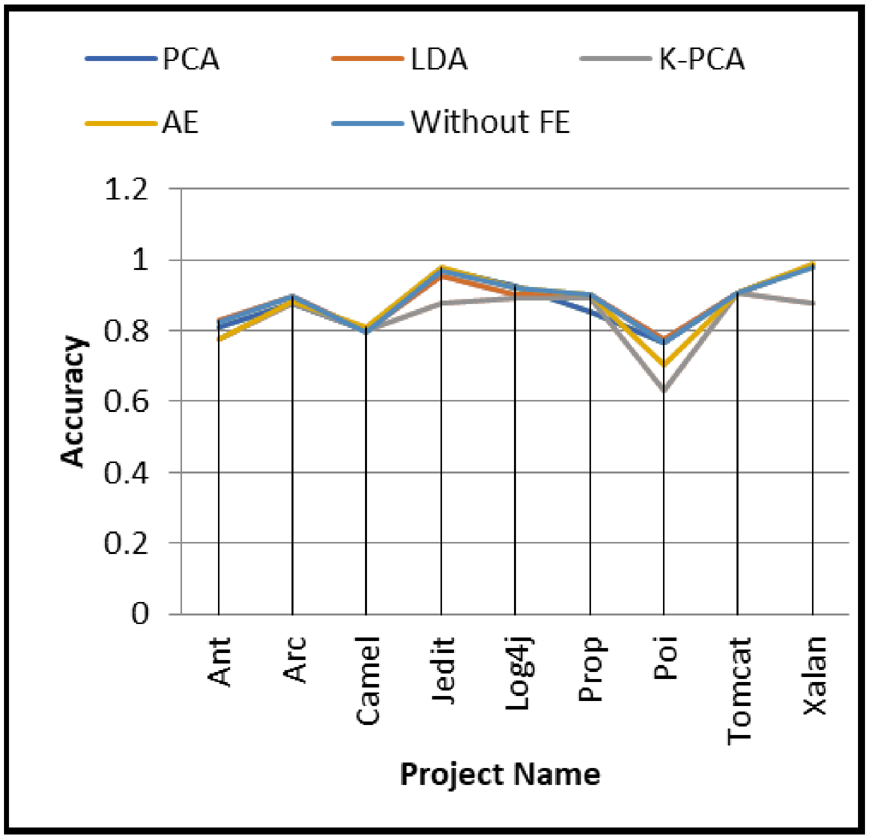


Figure 4.1: Accuracy of each FE technique.

4.4.2 Results Analysis based on RQ2

RQ2:To what extent do feature extraction methods influence the ROC-AUC performance of Support Vector Machine (SVM) models, and how do these methods compare when applied to datasets with diverse characteristics?

The evaluation of four feature extraction methods, as presented in Table 4.2 and Figure 4.2, reveals notable differences in their effectiveness, particularly highlighting the performance of Autoencoder (AE). With a mean accuracy of approximately 0.705, AE emerges as the top-performing technique, demonstrating its ability to capture complex patterns and relationships within the data. This high performance suggests that AE is particularly well-suited for feature extraction in contexts where capturing nuanced information is crucial for model accuracy.

In contrast, the approach of using raw features without any extraction also yields a commendable mean accuracy of around 0.693, placing it second overall. This finding is somewhat surprising, as it indicates that, for certain datasets, the inherent features may already contain sufficient information for effective predictive modeling. This underscores the importance of thoroughly understanding the dataset before opting for more complex feature extraction techniques.

Principal Component Analysis (PCA) ranks third with a mean accuracy of approximately 0.608. While PCA is widely recognized for its utility in reducing dimensionality, its performance in this analysis suggests that it may not capture all relevant aspects of the data as effectively as AE. Kernel PCA (K-PCA), which follows closely with a mean accuracy of about 0.595, indicates that while it is designed to model non-linear relationships, its performance does not significantly surpass that of PCA in this context.

Linear Discriminant Analysis (LDA) ranks lowest, with a mean accuracy of around 0.582. This suggests that LDA may not be suitable for datasets where class separation is not pronounced or where the assumptions of normality and equal variance do not hold, further emphasizing the need to align the choice of feature extraction method with the characteristics of the dataset.

The variability in performance across projects, ranging from 0.439 to 0.824, highlights the significance of project-specific characteristics in determining the effectiveness of feature extraction methods. This variability suggests the potential for tailoring methods to individual datasets or combining different techniques in ensemble approaches, thereby improving predictive performance in future applications.

Table 4.2: ROC-AUC Value for Each Project

Projects	PCA	LDA	K-PCA	AE	Without FE
Ant	0.817	0.840	0.773	0.818	0.792
Arc	0.499	0.838	0.467	0.738	0.735
Camel	0.625	0.435	0.515	0.696	0.689
Jedit	0.289	0.595	0.447	0.762	0.560
Log4j	0.704	0.253	0.424	0.745	0.716
Prop	0.345	0.360	0.579	0.508	0.401
Poi	0.810	0.841	0.807	0.847	0.816
Tomcat	0.592	0.595	0.625	0.757	0.707
Xalan	0.788	0.478	0.719	0.476	0.822

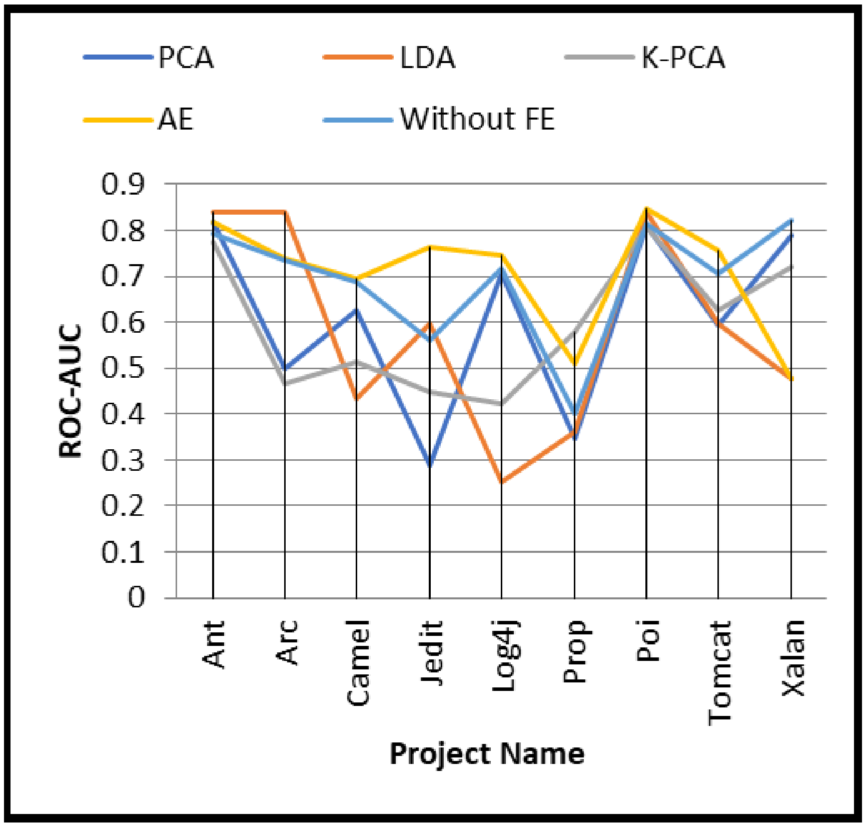


Figure 4.2: ROC-AUC of each FE technique.

4.4.3 Result Analysis based on RQ3

RQ3: What is the statistical significance of the differences in ROC-AUC performance among software defect prediction models using various feature extraction methods (Autoencoders, PCA, LDA, K-PCA), and how do these methods compare in their effectiveness?

Using a non-parametric Friedman test to evaluate the hypothesis, the results led to the rejection of the null hypothesis, indicating that the performance of models based on different feature extraction methods significantly differed. Notably, the Autoencoder method demonstrated superior performance, achieving the highest ROC-AUC score and ranking first among all techniques with a p-value of 0.009. This statistical significance suggests that Autoencoders are particularly effective for software defect prediction in this context. The results highlight the importance of selecting an appropriate feature extraction method, as the choice directly influences the model's performance. The significant differences in ROC-AUC scores between the various methods underscore the potential benefits of using advanced techniques like Autoencoders, which may capture complex data patterns more effectively than traditional methods such as PCA, LDA, or K-PCA. Overall, the findings emphasize the necessity for careful consideration of feature extraction methods in developing effective predictive models for software defects

4.5 Discussion

In this chapter, a comparison is performed on nine open-source software-systems written in Java from PROMISE Repository using four mostly used feature extraction technique such as Principal Component Analysis, Linear Discriminant Analysis, Kernel-based Principal Component Analysis and Autoencoders and a machine learning classifier, i.e. Support Vector Machine. The model validation is performed by 10-fold cross_ validation method, and the performance of the model is evaluated using accuracy and ROC-AUC. Results of this study indicate that Autoencoders is the effective method to reduce the dimensions of a software defect dataset successfully.

The findings highlight the effectiveness of using feature extraction techniques in conjunction with SVM for improving prediction accuracy across different software projects. PCA and AE appear to be the most promising techniques for enhancing SVM performance, while LDA shows strong potential in specific scenarios on the basis of accuracy as performance measure. Further exploration of these methods may lead to improved predictive modelling in software defect prediction.

The result analysis on the basis of ROC-AUC reveals that Autoencoder (AE) is the most effective feature extraction method, outperforming others with a mean accuracy of 0.705. Surprisingly, raw features are nearly as effective. The variability in performance across projects emphasizes the need for tailored approaches, suggesting that combining methods or customizing feature extraction techniques could enhance predictive modelling outcomes.

The analysis confirms significant differences in model performance based on feature extraction methods, with Autoencoders achieving the highest ROC-AUC score (p-value of 0.009). This underscores the effectiveness of Autoencoders for software defect prediction, highlighting the critical role of selecting suitable feature extraction techniques to enhance predictive model performance. In conclusion, this chapter underscores the pivotal role of feature extraction techniques in enhancing the predictive accuracy of software defect models. The comparative analysis of nine open-source software systems demonstrates that Autoencoders stand out as the most effective method, achieving the highest ROC-AUC scores. The findings advocate for the strategic selection and customization of feature extraction methods to optimize performance, revealing that a tailored approach can significantly impact the effectiveness of defect prediction models. Future research should continue to explore these techniques, aiming for innovations that further improve the reliability and accuracy of software defect predictions across diverse projects.

Chapter 5

Effect of Feature Selection on Cross-Project Defect Prediction

5.1 Introduction

For SDP usually we perform conventional Within-Project Defect Prediction (WPDP), in which for building the prediction model the training and test set belongs to a single project. But it becomes difficult to predict defects in a new project because of unavailability of past data and here WPDP remains ineffective.

However, CPDP in software has appealed much consideration of researchers in the area of software engineering. To build an effective classification model in case of unavailability of past data, CPDP turns out to be a promising method. In CPDP, the training of the model is performed on different project and testing is performed on the new project in which we intent to find the defects [153]. The dataset required for training & testing consists of different types of software metrics such as size metric, object- oriented metrics and process metrics. In many related works, researcher uses datasets consisting of combination of these metrics which leads to high dimensionality of datasets and unwittingly incorporates redundant and irrelevant software metrics into the dataset.

The predictive performance of classifiers in the CPDP model is adversely affected due to presence of redundant and irrelevant metrics [154]. These undesirable software metrics cause noise, over fitting and decrease in the performance of CPDP model. To deal with these serious issues many feature selection techniques have been employed for CPDP [155, 156]. These procedures decrease the dimensionality of dataset by removing the software metrics which are irrelevant to the target variable. In many related study authors have applied feature subset selection and feature ranking algorithms for CPDP. Some previous studies have proposed models for analysing the influence of FS techniques on the performance of cross-project defect prediction [157]. These methods for selecting features recognise and obtain the most useful features of the dataset that are essential for prediction process. Features can also play a significant role in a successful defect prediction model. If the data is available, but it is noisy, or the features are trivial, the model will produce an ineffective result. As a result, this data must undergo pre-processing to ensure that it is free of noise and is lesser redundant [158].

Ever since several software metrics have been used and a number of FS methods are existing to choose the most suitable metrics for CPDP, it is essential to investigate and compare them. This finding persuades us to perform an empirical study by comparing of five feature subset selection (FSS) and five feature ranking (FR) techniques for CPDP. The experiments are performed on four datasets that we have collected from open-source PROMISE repository. The dataset is pre-processed using SMOTE algorithm. Receiver Operating Characteristics- Area under Curve (ROC-AUC) is used as performance indicator along with statistical tests for evaluation. The aim of conducting this study is many-fold.

In this work, we aim to provide answers to the subsequent research questions

(RQs): The motivation behind each research question (RQ) is driven by the need to address critical challenges in Cross-Project Defect Prediction (CPDP) and the importance of feature selection techniques in improving prediction accuracy.

- RQ1: Which features are selected by FSS and feature ranking (FR) techniques for CPDP?

  This question aims to identify the specific software metrics selected by feature subset selection (FSS) and feature ranking (FR) techniques, which are essential to determine which features contribute most effectively to defect prediction. By exploring this, we aim to reduce the impact of irrelevant and redundant metrics, which can degrade model performance in CPDP.

- RQ2: Which FS technique performed best for CPDP?

  The motivation behind this question is to compare the effectiveness of different feature selection (FS) techniques, specifically FSS and FR methods, in terms of their ability to reduce dimensionality and improve the predictive performance of CPDP models. It is crucial to identify the best-performing FS technique, as the right method will enhance model accuracy and robustness by eliminating irrelevant metrics.

- RQ3: Which classifier performed best using FSS & FR techniques for CPDP?

  This question seeks to investigate how different classifiers perform when applied to datasets processed using FSS and FR techniques. The goal is to identify which classifier achieves the best predictive performance, taking into account the feature selection process, and ultimately enhance the effectiveness of CPDP models for defect prediction in new projects.

The organization of the chapter includes the Research Methodology in Section 5.2 beginning with data collection, it identifies the dependent and independent

variables, discusses data balancing techniques, and outlines the feature selection methods employed. The answers to the RQ's are reported in Section 5.3 followed by a discussion in Section 5.4. The results of the chapter are communicated as [CPDP paper].

5.2 Research Methodology

In this segment, brief information related to the dataset used, feature selection techniques, machine learning techniques, performance indicators, and statistical tests has been discussed.

5.2.1 Data Collection

We selected open-source publicly available Ant-(1.7), Camel-(1.6), Ivy-(2), and Tomcat-(6.0) datasets from the PROMISE repository. For a detailed description, Section 2.7 can be referred to. These datasets are among the most extensively used in the area of software defect prediction (SDP). The dataset consists of numeric data types only. The selection of PROMISE datasets was based on the size of the projects and the programming language used. These projects have more than 350 classes and several release versions. Hence, these Java-based projects were found to be most suitable for studies based on object-oriented metrics.

5.2.2 Dependent and Independent Variables

The independent variables used in this study are 20 object-oriented software metrics: WMC, DIT, NOC, CBO, RFC, LCOM, CA, CE, NPM, DAM, MOA, MFA, CAM, IC, CBM, AMC, MAX_CC, AVG_CC, LOC, and LCOM3. A complete overview can be found in Section 2.5.1 of Chapter 2 . The dependent variable (DV) is the target variable being measured in the experimental process. The DV used in this study is *defect_proneness*, which has a binary value of ‘0‘ or ‘1‘. A value of ‘0‘ indicates that the class or module is not prone to defects, whereas a value of ‘1‘ indicates defect-proneness.

5.2.3 Feature Selection Techniques

In this study, we utilized five wrapper-based techniques: Best First (BF), Exhaustive Search (ES), Genetic Search (GS), Greedy Stepwise Search (GSS), and Random Search (RS), as well as five filter-based techniques: Gain Ratio (GR), Symmetrical Uncertainty (SU), One-rule (OneR), Information Gain (IG), and ReliefF. These techniques were employed for feature selection (FS) to preprocess the datasets. A thorough overview is available in Section 2.9.

To apply the FS techniques to the datasets, a given model of FS for cross-project defect prediction (CPDP) is shown in Figure 5.1. Here, S represents the original source project containing a feature set $\{f_1, f_2, f_3, \ldots, f_m\}$. T is the original target project containing the same feature set. After performing feature selection on the source project S, we obtain S_p, a preprocessed source project containing a reduced set of features $\{f_1, f_2, f_3, \ldots, f_n\}$, where $n < m$. This reduced set of features is then applied to the target project T, yielding T_p, the preprocessed target project containing the reduced feature set. The datasets S_p and T_p are then used for training and testing, respectively, to calculate the results of CPDP with FS.

5.2.4 Data Balancing

The preprocessed dataset is balanced using SMOTE (Synthetic Minority Oversampling Technique) to address the issue of imbalanced class data. For a complete overview, refer to Chapter 2 Section. Many studies in existing SDP literature involve data

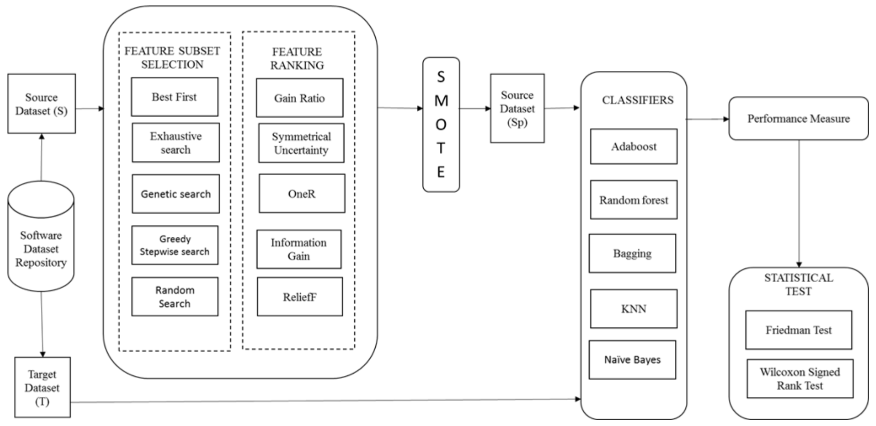


Figure 5.1: Framework for Feature Selection in Cross-Project Defect Prediction.

preprocessing by applying balancing techniques to improve results. SMOTE and its modified versions are the most popular and widely used methods for this purpose.

5.2.5 Machine Learning Classifiers

The attribute sets selected by each feature selection approach were evaluated using specific learning algorithms to compare the effectiveness of various feature selection strategies. The chosen algorithms include Naive Bayes (NB) for probabilistic classification, K-Nearest Neighbors (KNN) for instance-based learning, Bagging (BAG) for ensemble averaging, Random Forest (RF) for robust decision tree aggregation, and AdaBoost (AB) for adaptive boosting. For an in-depth overview of machine learning techniques, see Section 2.6.

In AdaBoost, the value of the *maximum_number_of_estimators* was set to ‘100‘, while all other parameters were left at their default values. Similarly, for Bagging, the *number_of_base_estimators* in the ensemble was set to ‘100‘, with other parameters at default settings. For Random Forest, all parameters were set to their default values. All experiments were performed using the Weka tool.

5.2.6 Performance Indicator

The Receiver Operating Characteristic - Area Under the Curve (ROC-AUC) was used as the performance indicator in our experiments to evaluate classifier performance. When dealing with the classification of imbalanced class data, conventional performance metrics such as accuracy, F-measure, or misclassification rate often prove ineffective. Section 2.12 contains a detailed explanation of ROC-AUC.

5.2.7 Statistical Test

To validate the implications derived from the ROC-AUC values, statistical validation was performed on the results. The Friedman test was used to determine whether significant differences existed among the performances of various classifiers. In cases where significant differences were observed, the Friedman test was followed by a post hoc Wilcoxon signed-rank test for pairwise comparisons. This provided a deeper understanding of classifier performance differences. A comprehensive overview of these statistical tests and their application in this context can be found in Section 2.13.

5.3 Result Analysis

The analysis of our results is structured around addressing the key research questions formulated at the outset of this study. Each research question was designed to evaluate specific aspects of the predictive modelling framework, feature selection techniques, and classifier performance in the context of software defect prediction. By aligning the results with these research questions, we have ensured a focused and systematic exploration of each objective.

5.3.1 Result Analysis based on RQ1

RQ1. Which features are selected by FSS and feature ranking FR techniques for CPDP?

To address RQ1, we analyzed the features selected by various feature subset selection (FSS) and feature ranking (FR) techniques across different datasets. Table 5.1 presents FSS techniquesâsuch as BF, ES, and RSâshowing that some techniques consistently select the same feature set (e.g., BF and GSS), while others (GS and ES) select a unique set for 50% of datasets. Table 5.2 provides details on FR techniques like GR, IG, and SU, which often share common features in certain datasets, while others (OneR and ReliefF) display more variability by selecting unique features each time. This comparison highlights how different techniques prioritize specific features across datasets, offering insight into their effectiveness for cross-project defect prediction.

Features Selected by FSS Techniques

Table 5.1 shows that FSS techniques BF and GSS always select the same set of features for all datasets. In contrast, the GS, ES, and RS techniques select a unique set of features for 50% of datasets, while for the other 50%, they select a common set of attributes.

Table 5.1: Details of selected features using the various FSS techniques.

Project	Technique	#Selected Features	Selected Features
Ant	BF, ES, GSS	2	RFC, NPM
	GS	3	CBO, RFC, NPM
	RS	3	RFC, CE, NPM
Camel	BF, GSS, RS	9	DIT, NOC, CBO, LCOM, CA, IC, CBM, AMC, MAX_CC
	ES	9	DIT, NOC, CBO, CA, NPM, IC, CBM, AMC, MAX_CC
	GS	10	WMC, DIT, NOC, CBO, CA, NPM, IC, CBM, AMC, MAX_CC
Ivy	BF, GSS	8	WMC, CBO, RFC, CE, NPM, LOC, MOA, AMC
	ES, GS, RS	9	CBO, RFC, LCOM, CE, NPM, LOC, DAM, MOA, AMC
Tomcat	BF, ES, GS, GSS	5	CBO, RFC, LOC, MOA, MAX_CC
	RS	4	CBO, RFC, LOC, MAX_CC

Features Selected by FR Techniques

Table 5.2 represents the software metrics (features) picked by the FR techniques based on the equation $\log_2(f)$, where $f = 20$ (the total number of features in each dataset). The initial four software metrics are chosen from the ranked list, as $\log_2(20) \approx 4$.

From the table, it can be determined that for three out of four datasets, GR and SU select the same set of features. In 50% of cases, IG selects common features with GR and SU, but in the remaining cases, it selects a unique set. In contrast, OneR and ReliefF consistently select a unique set of features.

Table 5.2: Details of selected features using the various FR techniques.

Project	**Technique**	**Selected Features**
Ant	GR, IG, SU	RFC, LOC, CAM, LCOM
	OneR	RFC, MAX_CC, CAM, LCOM
	ReliefF	RFC, MAX_CC, CE, LOC
Camel	GR	DIT, NOC, CA, CBM
	IG	CBO, CA, AMC, MAX_CC
	OneR	CA, DAM, IC, CBM
	ReliefF	DIT, CAM, IC, CBM
	SU	NOC, CBO, LCOM, CA
Ivy	GR, SU	MOA, LOC, RFC, NPM
	IG	LOC, RFC, WMC, LCOM
	OneR	CAM, CBO, LCOM, MOA
	ReliefF	DAM, LCOM3, CAM, MFA
Tomcat	GR, IG, SU	RFC, LOC, MAX_CC, CBO
	OneR	RFC, CE, DAM, NPM
	ReliefF	DAM, LCOM3, CAM, MFA

5.3.2 Result Analysis based on RQ2

RQ2. Which FS technique performed best for CPDP?

In Tables 5.3 to 5.6, we presented the ROC-AUC values of the five machine learning classifiers with the features selected by the FSS techniques, FR techniques, and when no feature selection was applied.

In Table 5.3, the values represent the ROC-AUC value for the case when Ant is taken as the source (training dataset) and Camel, Ivy, or Tomcat is taken as the target (testing dataset). Here, the ROC-AUC values vary from 0.53 to 0.82.For the case of Ant $\rightarrow$ Camel (where Ant is the training dataset and Camel is the testing dataset), the best value of 0.7 is achieved by two combinations: GS+AB and RS+RF, where GS and RS are the feature selection (FS) techniques and AB and RF are the classifiers. When Ant is the training dataset and Ivy is the testing dataset, the best value of 0.81 is given by the ReliefF+BAG combination. For Ant â Tomcat, the most promising ROC-AUC values are given by ReliefF+BAG, while the worst performance is observed with the GS and BAG combination.

When considering the case of Camel as the source dataset and the rest as the target datasets, as shown in Table 5.4, the ROC-AUC values range from 0.43 to 0.81. The best performance is achieved with the BAG classifier (0.81) using a subset of features selected by the GS algorithm. In contrast, the FR techniques do not perform well in this scenario, resulting in the least values (0.43), which is observed with the NB classifier when features are ranked using the OneR technique.

The ranked features did not substantially increase the ROC-AUC values compared to the values calculated for the entire set of features for the Ivy dataset. From Table 5.5, it is evident that the AB classifier provided the best value of 0.8 for Ivy when features were selected using the ES, GS, and RS techniques. The ROC-AUC values for the Ivy dataset range from 0.51 to 0.8.

Table 5.3: ROC-AUC Values for Ant

Category	**Types**	**Classifiers**	**Ant → Camel**	**Ant → Ivy**	**Ant → Tomcat**
	NONE	AB	0.67	0.81	0.76
		RF	0.56	0.73	0.75
No Feature Selection		BAG	0.63	0.80	0.81
		KNN	0.55	0.72	0.73
		NB	0.53	0.67	0.71
	BF, ES, GSS	AB	0.65	0.80	0.76
		RF	0.61	0.78	0.76
		BAG	0.68	0.79	0.77
		KNN	0.65	0.73	0.74
		NB	0.57	0.68	0.73
	GS	AB	0.70	0.79	0.76
		RF	0.58	0.75	0.75
Feature Subset Selection		BAG	0.53	0.71	0.67
		KNN	0.54	0.75	0.73
		NB	0.56	0.68	0.71
	RS	AB	0.60	0.80	0.74
		RF	0.70	0.75	0.70
		BAG	0.53	0.72	0.73
		KNN	0.55	0.74	0.70
		NB	0.53	0.67	0.72
	GR, IG, SU	AB	0.55	0.78	0.80
		RF	0.53	0.71	0.76
		BAG	0.54	0.73	0.75
		KNN	0.58	0.74	0.74
		NB	0.54	0.69	0.70
	OneR	AB	0.67	0.80	0.81
		RF	0.55	0.78	0.76
Feature Ranking		BAG	0.63	0.80	0.78
		KNN	0.55	0.71	0.71
		NB	0.54	0.67	0.72
	ReliefF	AB	0.66	0.78	0.82
		RF	0.58	0.76	0.76
		BAG	0.59	0.82	0.74
		KNN	0.55	0.77	0.74
		NB	0.54	0.69	0.73

As shown in Table 5.6, for the Tomcat dataset, the AB classifier consistently yields the maximum ROC-AUC value of 0.83, irrespective of the FS techniques used. The minimum ROC-AUC values of 0.53 are primarily observed with the NB classifier when features are ranked using RS, GR, IG, or SU techniques. Additionally, the minimum value is also observed with the KNN and RF classifiers when features are

Table 5.4: ROC-AUC Values for Camel

Category	**Types**	**Classifiers**	**Camel → Ant**	**Camel → Ivy**	**Camel → Tomcat**
No Feature Selection	NONE	AB	0.74	0.71	0.80
		RF	0.70	0.67	0.72
		BAG	0.74	0.75	0.75
		KNN	0.57	0.58	0.66
		NB	0.68	0.67	0.74
Feature Subset Selection	BF, GSS, RS	AB	0.73	0.76	0.79
		RF	0.69	0.71	0.76
		BAG	0.69	0.76	0.77
		KNN	0.56	0.59	0.60
		NB	0.56	0.62	0.69
	ES	AB	0.73	0.76	0.80
		RF	0.65	0.71	0.76
		BAG	0.68	0.78	0.76
		KNN	0.59	0.59	0.67
		NB	0.61	0.60	0.67
	GS	AB	0.74	0.73	0.77
		RF	0.67	0.76	0.74
		BAG	0.68	0.76	0.81
		KNN	0.65	0.60	0.64
		NB	0.58	0.63	0.69
Feature Ranking	GR	AB	0.62	0.56	0.59
		RF	0.59	0.60	0.64
		BAG	0.58	0.63	0.64
		KNN	0.57	0.55	0.59
		NB	0.51	0.52	0.54
	IG	AB	0.72	0.75	0.78
		RF	0.68	0.72	0.73
		BAG	0.65	0.70	0.77
		KNN	0.51	0.52	0.65
		NB	0.62	0.57	0.70
	OneR	AB	0.70	0.68	0.69
		RF	0.56	0.65	0.63
		BAG	0.59	0.63	0.61
		KNN	0.54	0.63	0.61
		NB	0.43	0.64	0.62
	ReliefF	AB	0.58	0.66	0.59
		RF	0.53	0.58	0.58
		BAG	0.58	0.63	0.59
		KNN	0.59	0.65	0.63
		NB	0.49	0.57	0.51
	SU	AB	0.60	0.63	0.60
		RF	0.55	0.59	0.60
		BAG	0.58	0.63	0.59
		KNN	0.60	0.65	0.61
		NB	0.48	0.53	0.51

Table 5.5: ROC-AUC Values for Ivy

Category	**Types**	**Classifiers**	**Ivy $\rightarrow$ Ant**	**Ivy $\rightarrow$ Camel**	**Ivy $\rightarrow$ Tomcat**
No Feature Selection	NONE	AB	0.66	0.59	0.67
		RF	0.76	0.55	0.67
		BAG	0.75	0.54	0.74
		KNN	0.73	0.52	0.72
		NB	0.71	0.52	0.74
Feature Subset Selection	BF, GSS	AB	0.75	0.62	0.75
		RF	0.75	0.57	0.73
		BAG	0.75	0.54	0.71
		KNN	0.75	0.53	0.74
		NB	0.67	0.53	0.72
	ES, GS, RS	AB	0.80	0.61	0.72
		RF	0.73	0.54	0.71
		BAG	0.74	0.55	0.72
		KNN	0.75	0.53	0.68
		NB	0.67	0.54	0.74
Feature Ranking	GR, SU	AB	0.77	0.66	0.77
		RF	0.71	0.54	0.69
		BAG	0.75	0.61	0.69
		KNN	0.74	0.53	0.70
		NB	0.67	0.53	0.74
	IG	AB	0.74	0.55	0.75
		RF	0.68	0.52	0.74
		BAG	0.70	0.54	0.72
		KNN	0.69	0.51	0.64
		NB	0.67	0.53	0.70
	OneR	AB	0.74	0.66	0.78
		RF	0.71	0.60	0.72
		BAG	0.69	0.62	0.77
		KNN	0.62	0.57	0.67
		NB	0.63	0.55	0.72
	ReliefFs	AB	0.70	0.54	0.72
		RF	0.66	0.54	0.71
		BAG	0.68	0.54	0.67
		KNN	0.68	0.55	0.65
		NB	0.70	0.55	0.76

ranked by the ReliefF algorithm.

To find the best performing feature selection technique independently, we calculated the average AUC-ROC values for each FSS and FR technique. From Table 5.7, it can be observed that BF, ES, and GSS perform the best with an approximate average

Table 5.6: ROC-AUC Values for Tomcat

Category	Types	Classifiers	Tomcat → Ant	Tomcat → Camel	Tomcat → Ivy
No Feature Selection	NONE	AB	0.81	0.68	0.83
		RF	0.69	0.62	0.74
		BAG	0.75	0.58	0.76
		KNN	0.64	0.53	0.67
		NB	0.65	0.55	0.68
Feature Subset Selection	BF, ES, GS, GSS	AB	0.78	0.70	0.83
		RF	0.72	0.67	0.73
		BAG	0.74	0.70	0.81
		KNN	0.65	0.65	0.78
		NB	0.64	0.59	0.69
	RS	AB	0.75	0.69	0.83
		RF	0.72	0.70	0.77
		BAG	0.76	0.70	0.81
		KNN	0.64	0.56	0.69
		NB	0.64	0.53	0.68
Feature Ranking	GR, IG, SU	AB	0.74	0.60	0.78
		RF	0.67	0.57	0.70
		BAG	0.73	0.60	0.77
		KNN	0.64	0.56	0.68
		NB	0.65	0.53	0.69
	OneR	AB	0.77	0.61	0.81
		RF	0.69	0.57	0.76
		BAG	0.70	0.61	0.75
		KNN	0.64	0.54	0.68
		NB	0.67	0.55	0.69
	ReliefF	AB	0.72	0.56	0.82
		RF	0.68	0.53	0.68
		BAG	0.65	0.53	0.67
		KNN	0.66	0.55	0.69
		NB	0.69	0.54	0.68

Table 5.7: Average AUC-ROC Values of FSS and FR Techniques

None	BF	ES	GSS	GS	RS	GR	IG	SU	OneR	ReliefF
0.68	0.70	0.70	0.70	0.68	0.68	0.65	0.66	0.65	0.66	0.64

AUC-ROC value of 0.70. Hence, it is evident that FSS techniques outperform FR techniques when considering the average value of 60 combinations of cross-project defect validation.

5.3.3 Result Analysis based on RQ3

RQ3. Which classifier performed best using FSS & FR techniques for cross-project defect prediction

Table 5.8: Mean Rank of the Classifiers Calculated by Friedman Test for FSS and FR Techniques

Classifiers	Mean Rank (Feature Subset Selection)	Mean Rank (Feature Ranking)
AB	1.56	1.51
RF	2.70	3.10
BAG	2.31	2.58
KNN	4.29	3.77
NB	4.14	4.04

From Table 5.8, the Friedman test results indicate that the AB classifier exhibited the best performance with mean ranks of 1.56 and 1.51 for FSS and FR techniques, respectively. The null hypothesis, which states that ”the predictive performance of all classifiers is equal,” is rejected, confirming statistically significant differences among the classifiers’ performances.

Table 5.9: Results of Wilcoxon Signed-Rank Test for Pairwise Comparison of Classifiers

Feature Subset Selection	AB	RF	BAG	KNN	NB	Feature Ranking	AB	RF	BAG	KNN
AB	*	S	S	S	S	AB	*	S	S	S
RF		*	S	S	S	RF		*	NS	S
BAG			*	S	S	BAG			*	S
KNN				*	NS	KNN				*
NB					*	NB				

Table 5.9 displays the results of the Wilcoxon signed-rank test. For FSS techniques, pairwise performances of all classifiers were significantly different (S), except for KNN and NB, where the null hypothesis was accepted. For FR techniques, two pairsâRF and BAG, and KNN and NBâdid not exhibit statistically significant differences (NS).

Based on these statistical tests, it can be concluded that: AB is the best-performing classifier for both FSS and FR techniques, showing statistically significant differences from other classifiers. KNN and NB were the least-performing classifiers, with no significant difference in their pairwise performances in specific cases.

5.4 Discussion

This chapter presents a comprehensive empirical comparison of feature selection (FS) techniques for cross-project defect prediction (CPDP) using publicly available PROMISE datasets. By examining five feature subset selection (FSS) and five feature ranking (FR) techniques, alongside five machine learning classifiers, we aim to assess the impact of FS on model performance.
Key Findings

- Both FSS and FR techniques significantly improve CPDP performance over using no feature selection, emphasizing the need for a representative feature subset or rationally ranked features to optimize model performance.
- When averaged across all datasets, FSS techniques consistently outperform FR methods and scenarios without FS, demonstrating the advantage of FSS in enhancing CPDP results.
- ROC-AUC values and statistical testing at a 95% confidence interval identify AdaBoost (AB) as the most effective classifier, frequently producing superior results when paired with FSS-selected features.

Findings Within Feature Selection Techniques

- BF and GSS consistently select the same set of features across datasets, indicating stability.

- GS, ES, and RS offer flexibility by selecting unique sets of features in half of the cases, and common features in the remaining cases.
- GR and SU typically select common feature sets across three out of four datasets.
- OneR and ReliefF regularly choose distinct feature sets, providing flexibility for feature selection based on specific dataset needs.

Conclusively, ensemble-based classifiers like AdaBoost, when paired with FSS-selected features, yield the most promising models for defect prediction, with this approach being highly applicable across similar projects. Future research will explore evolutionary FS techniques and expand CPDP analyses to include additional open-source software projects, aiming to refine and generalize these findings further.

Chapter 6

A Novel Software Defect Prediction Model using two-phase Grey Wolf Optimisation for Feature Selection

6.1 Introduction

Software quality is a crucial factor in software engineering, directly influencing user satisfaction, business efficiency, and cost-effectiveness. High-quality software, with fewer defects, improves performance, reduces maintenance efforts, and offers a competitive edge in the marketplace. Software Defect Prediction (SDP) helps identify code sections prone to defects, improving development by targeting high-risk areas. Feature Selection (FS) in SDP enhances model performance by eliminating irrelevant or redundant features, thereby reducing dimensionality and improving classification accuracy [159].

Evolutionary FS techniques, such as Grey Wolf Optimizer (GWO), have emerged as effective solutions to address the challenges of high-dimensional datasets. These techniques, particularly metaheuristic-based approaches, help optimize the feature subset by balancing exploration and exploitation. However, traditional methods like exhaustive search are computationally expensive, making evolutionary methods like GWO more suitable for large datasets. By using natural-inspired algorithms, these techniques overcome local optima and enhance diversity, improving feature selection efficiency [160–165].

In this chapter, we propose a novel SDP model using the 2M-GWO algorithm, which incorporates a two-phase mutation operation to improve performance and avoid local optima. The model was tested on 27 publicly available datasets, with results demonstrating its superior performance in selecting the most relevant features. Comparisons with other metaheuristic techniques, such as Harris Hawks Optimization and Whale Optimization, showed the 2M-GWO's robustness and effectiveness in optimizing SDP tasks. The statistical analysis using ROC-AUC values further validated the modelâs performance. This approach represents a significant advancement in SDP, showcasing the potential of evolutionary FS techniques in improving the accuracy and efficiency of software defect prediction. The chapter explores the following RQs:

RQ 1. What is the effect of FS techniques on the SDP models? Does 2M-GWO significantly improve the performance of SDP models as compared to GWO and WFS Models?

The primary motivation behind RQ 1 is to investigate the impact of Feature Selection (FS) techniques on the performance of Software Defect Prediction (SDP) models. FS techniques are crucial for reducing dimensionality, removing irrelevant features, and improving the generalization ability of predictive models. By exploring the effect of FS techniques, specifically the Grey Wolf Optimizer (GWO) and the two-phase mutation GWO (2M-GWO), this research aims to evaluate whether the incorporation of 2M-GWO results in a significant improvement in the prediction accuracy compared to traditional methods. Understanding this relationship can lead to more efficient and accurate defect prediction models.

RQ 2 Which is the best-performing ML technique for predicting software defects?

RQ 2 seeks to identify the best-performing machine learning (ML) technique for predicting software defects. Different ML classifiers exhibit varying levels of performance depending on the feature set and dataset used. The motivation is to compare the predictive power of various ML techniques in the context of software defect prediction. By determining which classifier consistently performs better, this research will provide valuable insights for practitioners in selecting the most effective ML techniques for defect prediction tasks, ultimately improving software quality.

RQ 3. Which software metrics are frequently selected using the proposed algorithm?

RQ 3 aims to explore which software metrics are most frequently selected by the proposed 2M-GWO algorithm. Feature selection is a crucial aspect of SDP, and the success of defect prediction depends on selecting the most relevant features. By understanding which metrics are consistently chosen, this research aims to reveal key software characteristics that are strong indicators of defects. This can help software engineers and researchers focus on the most important metrics, improving both the prediction model and the software development process.

RQ 4 What is the predictive performance of the proposed model as compared to other benchmark FS models for SDP?

The motivation for RQ 4 is to evaluate the performance of the proposed 2M-GWO-based SDP model against other benchmark feature selection (FS) models. By comparing the predictive accuracy of the proposed model with established FS techniques, this research will assess the effectiveness and competitiveness of the 2M-GWO algorithm in defect prediction. This comparison will highlight the advantages or limitations of the proposed approach, providing evidence of its superiority or suggesting areas for further improvement in the field of software defect prediction.

These RQ’s statistically compares the performance of the models developed, providing insights into the efficacy of various feature selection techniques and machine learning classifiers in software defect prediction. By identifying key metrics and evaluating predictive capabilities, this study contributes valuable knowledge to enhance software quality and reliability in development practices.

This chapter details the proposed methodology in section 6.2, starting with the dataset selection and feature selection process using the 2M-GWO algorithm, including the Grey Wolf Optimizer and its two-phase mutation variant. It then discusses classifiers, evaluation methods, parameter settings, and statistical tests. Finally, Section 6.3 presents the result analysis and Section 6.4 discusses the key findings. The result of the chapter is communicated as [166]

6.2 Proposed Methodology

To build an efficient Software Defect Prediction (SDP) model, we developed a wrapper strategy [70] based on an enhanced version of the Grey Wolf Optimizer (GWO) algorithm. Figure 6.1 explores the mechanism of the suggested algorithm. A 10-fold cross-validation (10-CV) procedure is performed at each iteration. Initially, the algorithm applies 2M-GWO on the SDP dataset. Using the 10-CV method, the chosen dataset is subdivided at each iteration. The SMOTE [167]mechanism is used to resolve the unbalanced problem of the dataset on which the model should be trained. This procedure will generate a training dataset using oversampling.

Five distinct classifiers (i.e., SVM, RF, GB, AB, and KNN) are used to construct an SDP model, which is then assessed using K-Fold = 10. The suggested method terminates whenever the best solution in terms of the objective function is attained, or the value of iterations reaches its maximum value.

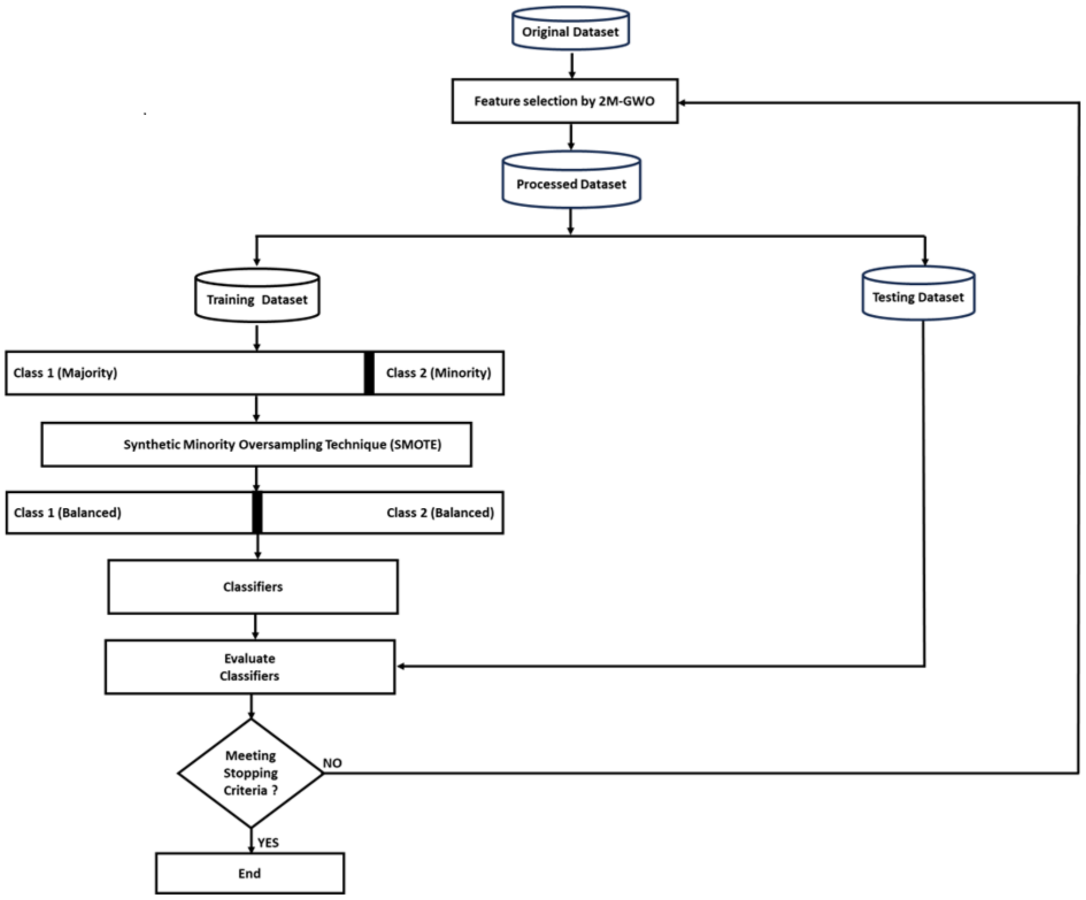


Figure 6.1: Mechanism of the suggested 2M-GWO algorithm.

6.2.1 Dataset Details

For this study's defect datasets, data from four different sources were collected (NASA, PROMISE, RELINK, and AEEEM). This study made use of the NASA dataset as it was created by Shepperd, et al. The NASA datasets include metrics generated from static code parts that take into account the size and complexity of the code. Projects from the PROMISE repository are based on features that are gathered at the class level for object-oriented software and information about defects found in software components. This dataset was developed using the JAVA programming language and was retrieved from the Apache software [106]. The open-source RELINK dataset comprises three open-source projects (Apache HTTP, Safe, and Zxing) and has a total of twenty-six attributes [52].

AEEEM datasets differ from other data [10] as they include software attributes from source code measurements. The change metrics, entropy, and churn of source code metrics form the foundation of source code metrics. The datasets include Ant-1.7, Camel-1.2, Camel-1.4, Jedit-3.2, Jedit-4.1, Jedit-4.2, Log4j-1.0, Log4j-1.1, Lucene-2.0, Lucene-2.2, Tomcat, Xalan-2.4, Xalan-2.5, Xalan-2.6, Apache, Safe, Zxing, Cm1, Mw1, PC1, PC3, PC4, EQ, JDT, LC, ML, and PDE. These datasets are obviously dependent on various software features. The datasets details are mentioned in Chapter 2.

6.2.2 Feature selection using 2M-GWO

Grey Wolf Algorithm (GWO)

Grey wolves (GWs) are superb predators with exceptional hunting abilities. This is due to the fact that they live in a disciplined and orderly pack. The GWO algorithm was recently presented to imitate the hunting, seeking, and surrounding behaviour of GWs [161]. There are four main varieties of GWs with varying levels of dominance and leadership in the wolf population. The first, second, and third-best solutions in the GWO method are denoted by the letters α, β, and δ, correspondingly. The rest of the possible solutions are referred to as ω.

The hunting mechanism in GWO involves three main steps: encircling the prey, hunting (attacking), and searching for prey. These steps are mathematically modelled as follows:

- *Encircling Prey:* The wolves encircle the prey during the hunt. This behaviour is modelled using the position vectors of the prey and the wolves. The position of each wolf is updated according to the position of the prey using the following equations:

$$D_v = |C_v \cdot X_v^p - X_v(t)| \tag{1}$$

$$X_v(t+1) = X_v^p(t) - A_v \cdot D_v \tag{2}$$

$$A_v = 2a_v \cdot \text{random}_{v1} - a_v \tag{3}$$

$$C_v = 2 \cdot \text{random}_{v2} \tag{4}$$

$$a = 2 - t \cdot \frac{2}{\text{iterations}} \tag{5}$$

Here, $X_v(t)$ represents the position vector of a wolf at iteration t, and $X_v^p(t)$ represents the position of the prey. C_v and A_v are coefficient vectors. The parameter *iterations* determines the maximum number of iterations. The random$_{v1}$ and random$_{v2}$ are random vectors, whereas a_v decreases linearly from 2 to 0 over the course of iterations.

- *Attacking the Prey:* The alpha (best solution), beta, and delta wolves have better knowledge about the potential location of the prey. Therefore, the positions of the other wolves (omegas) are updated based on the positions of the alpha, beta, and delta wolves:

$$X_v(t+1) = \frac{X_{v1} + X_{v2} + X_{v3}}{3} \tag{6}$$

Here, X_{v1}, X_{v2}, and X_{v3} are the positions of the alpha, beta, and delta wolves, respectively, updated using similar equations to the encircling mechanism but centered around each of these three wolves.

- *Searching for Prey:* To simulate the search for prey, the wolves diverge from each other and explore the search space. This is modelled by allowing the random vectors random$_{v1}$ and random$_{v2}$ to take values that can potentially increase the distance between the position of a wolf and the position of the prey.

Although the standard GWO is known for its effectiveness in various optimization tasks, but it faces challenges such as premature convergence and getting trapped in local optima when dealing with complex feature selection problems. To address these challenges, the authors propose a two-phase mutation mechanism integrated into the GWO. This enhancement aims to improve the exploitation capabilities of the algorithm, making it more effective in navigating the search space of feature selection problems.

Two-phase mutation Grey Wolf Optimisation (2M-GWO)

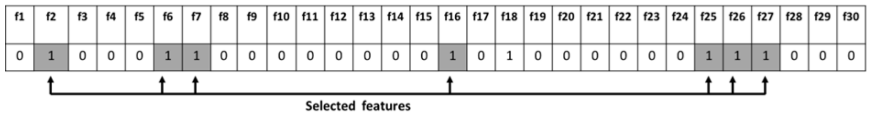
The Two-Phase Mutation Grey Wolf Optimizer (TMGWO) is an enhanced version of the standard GWO, specifically adapted for feature selection problems in classification tasks. This enhancement integrates a two-phase mutation mechanism to improve the exploitation capabilities of the algorithm, making it more effective in navigating the complex search space of feature selection problems. The benefits of using two mutation phases are as follows:

First Mutation Phase: This phase aims to reduce the number of selected features while maintaining high classification accuracy. It selectively mutates certain features of the best solution (alpha wolf), turning some of the selected features off (from 1 to 0).

Second Mutation Phase: This phase attempts to add potentially informative features that could increase classification accuracy. It operates by turning some unselected features on (from 0 to 1) based on their potential to improve the solution.

This 2M-GWO algorithm includes four steps: Initialization (step 1), evaluation (step 2), transformation function (step 3), and two-phase mutation (step 4) make up the core components of 2M-GWO.

- **Initilisation**:The first step is to initialize a colony of n grey wolves (GWs) or search agents (SA) generated randomly. Each SA represents a potential solution and has a size of d, equal to the number of attributes in the actual dataset [167]. For classifying classes, the feature selection (FS) process is used to identify the subset of features that produce the most accurate results. This is achieved by setting some features to ”1” (selected) and others to ”0” (not selected), as shown in Fig. 6.2. At the start, each solution is represented by binary values of either ”0” or ”1”.



f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13	f14	f15	f16	f17	f18	f19	f20	f21	f22	f23	f24	f25	f26	f27	f28	f29	f30
0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	1	0	0	0

Figure 6.2: Solution representation of 2M-GWO.

- **Evaluation**

The feature selection (FS) process has multiple objectives, as it must achieve two goals: reduce the count of features selected and increase the classifier’s accuracy. To achieve this balance, the fitness function is designed to evaluate the solutions and meet both objectives simultaneously, as given in Eq.7.

$$\text{fitness} = \alpha \cdot \gamma_R(D) + \beta \cdot \frac{|s|}{|d|} \tag{7}$$

Here, $\gamma_R(D)$ represents the rate of classification error of R, the condition attribute set, with respect to decision D, which is computed via the machine learning (ML) classifier KNN. The span of the chosen attribute subset’s cardinality is represented by $|s|$. Each sample of the actual dataset that contains features is represented by a cardinality of $|d|$. α and β are weight constraints, which are equivalent to the values of the classification accuracy and the length of the subset of the chosen feature, respectively. These parameters satisfy

$\alpha \in [0, 1]$ and $\alpha + \beta = 1$.

The evaluation function will overlook results that might have equivalent accuracy but use fewer attributes, which is a key component in reducing the dimensionality problem if it solely considers classification accuracy.

- **Transformation Function**

  The grey wolf's (GW) positions produced by the typical GWO are continuous in nature. Since this conflicts with the binary structure of the feature selection (FS), it cannot be directly applied to this case. The issue with FS arises from selecting (1) or ignoring (0) the advantageous attributes that improve the classifier's capability. The task of converting the continuous values of the search space into binary values is performed by the transformation function (TF). The TF used in this study is an S-shaped sigmoidal function [168].

  The conversion of continuous values to binary values is achieved using the following equations of the sigmoidal transformation function:

$$x_i = \frac{1}{1 + e^{-x_i}} \tag{8}$$

$$x_{\text{binary}} = \begin{cases} 0 & \text{if rand} < x_i \\ 1 & \text{if rand} \geq x_i \end{cases} \tag{9}$$

  Here, x_i represents the continuous value of the feature, where i varies from 1 to d. x_{binary} can have a value of 0 (not selected) or 1 (selected) based on comparing x_i with a random value *rand*.

- **Two-Phase Mutation**

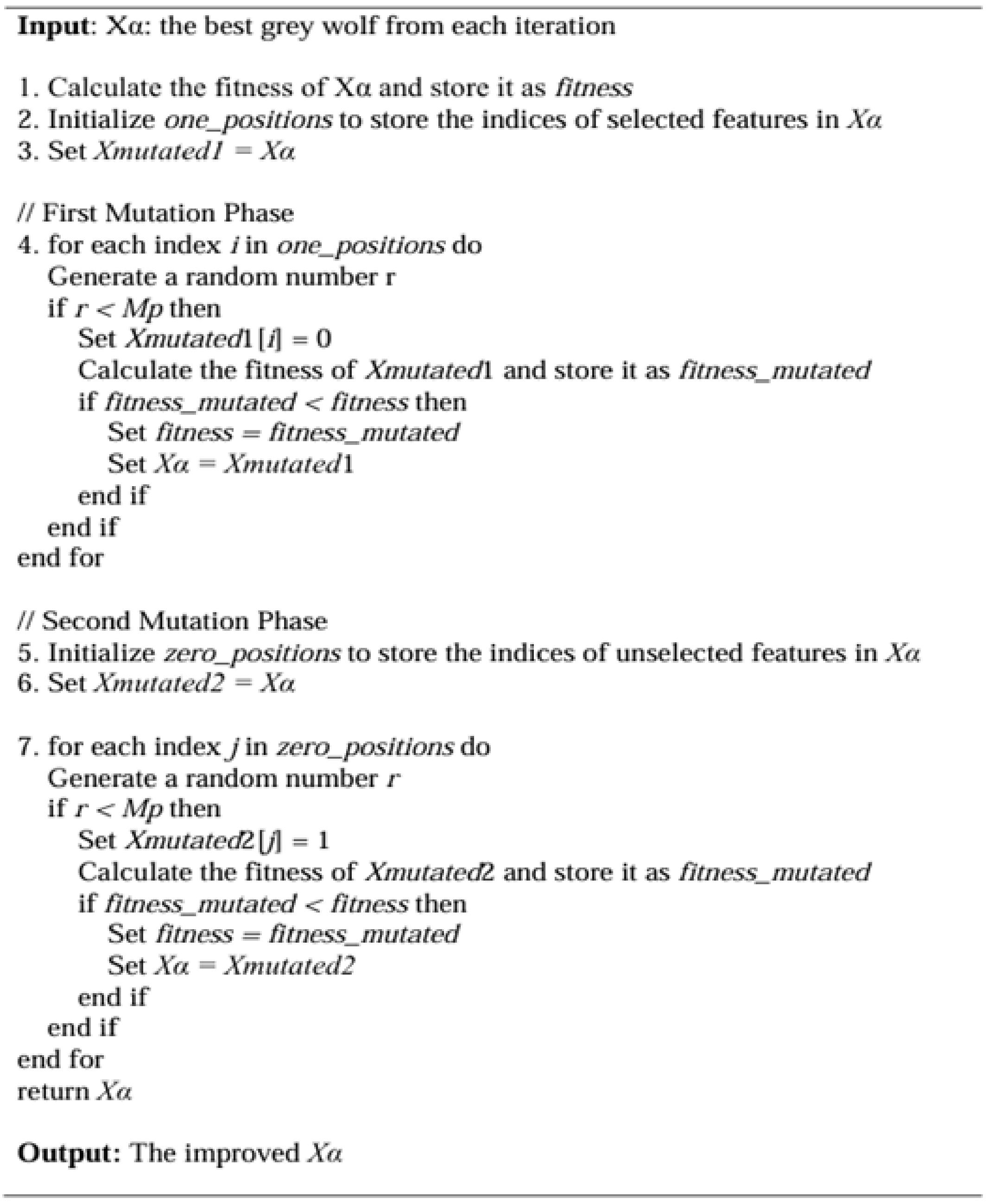
A mutation operator is utilized to enhance the performance of the proposed algorithm. In the initial phase of mutation, the primary objective is to reduce the number of selected attributes while preserving high classification accuracy [168]. The subsequent phase focuses on improving classification efficacy by introducing more nuanced attributes. To mitigate delays that may arise from the mutation operator, step 4 can be executed with a mutation probability (M_p). This approach optimizes the mutation process, allowing for controlled adjustments without compromising efficiency.

Figure 6.3 provides the detailed pseudocode for the proposed mutation process. Additionally, the complete pseudocode for the 2M-GWO algorithm, incorporating both mutation and other key components, is outlined in Algorithm 2.

In Figure 6.4, X_α, X_β, and X_δ represent the positions of the best three grey wolves (alpha, beta, delta), which guide the search process. X_i denotes the position of the i-th grey wolf. *Fitness* is the variable that shows the fitness value of X_α. *one_positions* is a vector storing the positions of selected features in X_α, and *zero_positions* stores the positions of unselected features in X_α.

```
Inputs: n, Number of grey wolves in the population.
iterations, maximum number of iterations.

1. Initialize a population of n grey wolves Xi, i = 1 to n
2. Initialize parameters a, A, and C
3. Calculate the fitness of each grey wolf
4. Assign the best three grey wolves to Xα, Xβ, and Xδ respectively
5. Set t = 0

6. while t < iterations do
    for each grey wolf Xi do
        Update the position of Xi using Eq. (6)
    end for
    Update parameters a, A, and C
    Convert the position of grey wolves to binary using Eq. (8) and Eq. (9)
    Calculate the fitness of all grey wolves
    Update the best three grey wolves Xα, Xβ, and Xδ
    Apply two-phase mutation to Xα and update its fitness
    Increment t by 1
end while

return the best grey wolf Xα

Output: The best grey wolf Xα
```

Figure 6.3: Pseudocode of the proposed 2M-GWO.



```
Input: Xα: the best grey wolf from each iteration

1. Calculate the fitness of Xα and store it as fitness
2. Initialize one_positions to store the indices of selected features in Xα
3. Set Xmutated1 = Xα

// First Mutation Phase
4. for each index i in one_positions do
   Generate a random number r
   if $r < Mp$ then
      Set Xmutated1 [i] = 0
      Calculate the fitness of Xmutated1 and store it as fitness_mutated
      if fitness_mutated < fitness then
         Set fitness = fitness_mutated
         Set Xα = Xmutated1
      end if
   end if
end for

// Second Mutation Phase
5. Initialize zero_positions to store the indices of unselected features in Xα
6. Set Xmutated2 = Xα

7. for each index j in zero_positions do
   Generate a random number r
   if $r < Mp$ then
      Set Xmutated2 [j] = 1
      Calculate the fitness of Xmutated2 and store it as fitness_mutated
      if fitness_mutated < fitness then
         Set fitness = fitness_mutated
         Set Xα = Xmutated2
      end if
   end if
end for
return Xα

Output: The improved Xα
```

Figure 6.4: Pseudocode of the Two-Phase Mutation step

6.2.3 Classifiers

The literature has a number of learning classifiers. As a result, we restrict our research to using only five distinct classifiers, namely SVM, RF, GB, AB, and KNN. The detailed overview of these techniques can be found in Chapter 2, Section.

6.2.4 Evaluation Method

To assess the model's performance and facilitate result comparison, we employ ROC_AUC. It is a graphical tool that assesses a binary classifier's effectiveness by varying the discrimination threshold value.

6.2.5 Parameter Setting

To address the stochastic character, each algorithm is performed 20 times independently using a random seed. The maximum number of iterations for all subsequent tests is set at 100. There are 5 search agents (SA) in the entire population. Additionally, 10-CV is used in this case.

In order to choose the optimal values for α and β, and based on certain values collected from past studies, several tests are carried out on various datasets. As a consequence, α and β are set to corresponding values of 0.01 and 0.99. To enable a fair comparison of the algorithms, the parameter specifications of the techniques are collected from existing studies. The parameters of some of the optimization techniques are mentioned in Table 6.1. For all other techniques, the default value is taken.

Table 6.1: Parameter settings for Optimization techniques

Technique	**Parameter**	**Value**
2M-GWO	Mutation Probability, M_p	0.5
SCO	Constant value, a	2
WO	Spiral Coefficient, b	1

6.2.6 Statistical Test

In software defect prediction, the Friedman test is often employed to evaluate and compare the effectiveness of various defect prediction models or techniques. This is crucial because software defect prediction models are designed to predict the likelihood of defects in software modules, and choosing the most effective model can significantly impact the quality and reliability of software.

6.3 Results Analysis

This subsection provides the findings in response to the Research Questions (RQ) as follows:

6.3.1 Result Analysis based on RQ1

RQ 1. What is the effect of FS techniques on the SDP models? Does 2M-GWO significantly improve the performance of SDP models as compared to GWO and WFS Models?

Table 6.2-7.3 explores the experimental results of the original GWO, modified 2M-GWO and WFS model for 27 datasets using five classifiers, namely SVM, RF, GB, AB and KNN. The experimental data include the feature size and ROC-AUC value for each dataset. We employed SMOTE to balance the classes in the given dataset. The ROC-AUC value for 27 open-source datasets is obtained by taking the average value of 20 runs for each project.

Table 6.2: ROC-AUC Scores by SVM

Projects	**GWO**	**2M-GWO**	**WFS**
PROMISE-Ant-1.7	0.821	0.815	0.880
PROMISE-Camel-1.2	0.515	0.618	0.618
PROMISE-Camel-1.4	0.704	0.629	0.629
PROMISE-Jedit-3.2	0.779	0.810	0.782
PROMISE-Jedit-4.1	0.870	0.783	0.725
PROMISE-Jedit-4.2	0.747	0.830	0.725
PROMISE-Log4j-1.0	0.912	0.914	0.900
PROMISE-Log4j-1.1	0.867	0.880	1.000
PROMISE-Lucene-2.0	0.765	0.807	0.807
PROMISE-Lucene-2.2	0.656	0.589	0.589
PROMISE-Tomcat	0.709	0.737	0.737
PROMISE-Xalan-2.4	0.757	0.672	0.672
PROMISE-Xalan-2.5	0.603	0.752	0.647
PROMISE-Xalan-2.6	0.649	0.961	0.752
AEEEM-EQ	0.801	0.803	0.825
AEEEM-JDT	0.836	0.848	0.556
AEEEM-LC	0.801	0.868	0.556
AEEEM-ML	0.772	0.798	0.556
AEEEM-PDE	0.653	0.755	0.667
NASA-Cm1	0.751	0.795	0.698
NASA-Mw1	0.578	0.777	0.683
NASA-PC1	0.736	0.846	0.849
NASA-PC3	0.812	0.896	0.852
NASA-PC4	0.851	0.906	0.919
RELINK-Apache	0.732	0.776	0.689
RELINK-Safe	0.629	0.629	0.886
RELINK-Zxing	0.667	0.714	0.705
Average (Avg)	**0.739**	**0.785**	**0.737**

The data presented in Table 6.2 clearly demonstrates that the 2M-GWO algorithm outperforms the other two algorithms, showing promising results in terms of ROC-AUC values across a majority of projects. Analysis of average ROC-AUC values indicates that the modified 2M-GWO algorithm surpasses the original GWO and WFS methods, with a maximum average value of 0.785. Furthermore, the proposed algorithm, in conjunction with SVM, significantly improved average ROC-AUC values by over 6% across all datasets, demonstrating strong overall performance.

Table 6.3 illustrates the results of the RF classifier utilized to evaluate the original GWO, the modified 2M-GWO, and the complete dataset. It is evident that the 2M-

Table 6.3: ROC_AUC Scores by RF

Projects	**GWO**	**2M-GWO**	**WFS**
PROMISE-Ant-1.7	0.832	0.845	0.843
PROMISE-Camel-1.2	0.641	0.634	0.652
PROMISE-Camel-1.4	0.736	0.710	0.715
PROMISE-Jedit-3.2	0.775	0.776	0.758
PROMISE-Jedit-4.1	0.886	0.862	0.692
PROMISE-Jedit-4.2	0.733	0.730	0.692
PROMISE-Log4j-1.0	0.879	0.889	0.871
PROMISE-Log4j-1.1	0.876	1.000	1.000
PROMISE-Lucene-2.0	0.730	0.767	0.749
PROMISE-Lucene-2.2	0.563	0.616	0.616
PROMISE-Tomcat	0.811	0.830	0.824
PROMISE-Xalan-2.4	0.754	0.749	0.715
PROMISE-Xalan-2.5	0.656	0.827	0.748
PROMISE-Xalan-2.6	0.820	1.000	0.814
AEEEM-EQ	0.818	0.841	0.825
AEEEM-JDT	0.877	0.881	0.850
AEEEM-LC	0.884	0.821	0.849
AEEEM-ML	0.859	0.838	0.857
AEEEM-PDE	0.787	0.784	0.793
NASA-Cm1	0.623	0.718	0.766
NASA-Mw1	0.733	0.500	0.613
NASA-PC1	0.927	0.925	0.930
NASA-PC3	0.825	0.835	0.827
NASA-PC4	0.919	0.931	0.938
RELINK-Apache	0.732	0.697	0.680
RELINK-Safe	0.786	0.786	0.971
RELINK-Zxing	0.611	0.827	0.807
Average (Avg)	**0.780**	**0.800**	**0.792**

GWO algorithm outperforms the other two methods, as it demonstrates the highest ROC-AUC values across a larger number of projects. The enhanced 2M-GWO surpasses the original GWO and WFS techniques in terms of average ROC-AUC values, with a maximum average value of 0.8. The average ROC-AUC value for all datasets saw an increase of approximately 2% when utilizing the proposed algorithm.

The ROC-AUC value calculated by Gradient Boosting (GB) for the full dataset and processed datasets, in which features were selected GWO and 2M-GWO, is presented in Table 6.4. The results indicate that the 2M-GWO-based model demonstrates superior performance compared to the GWO and WFS cases, with an average improvement of 2.3% in the ROC-AUC value. In approximately 40% of datasets, the 2M-GWO-based model outperformed the others, while in approximately 19% of

Table 6.4: ROC-AUC Scores by GB

Projects	**GWO**	**2M-GWO**	**WFS**
PROMISE-Ant-1.7	0.780	0.710	0.710
PROMISE-Camel-1.2	0.514	0.551	0.551
PROMISE-Camel-1.4	0.726	0.673	0.673
PROMISE-Jedit-3.2	0.823	0.771	0.771
PROMISE-Jedit-4.1	0.786	0.819	0.702
PROMISE-Jedit-4.2	0.687	0.731	0.702
PROMISE-Log4j-1.0	0.700	0.800	0.764
PROMISE-Log4j-1.1	0.752	0.933	0.933
PROMISE-Lucene-2.0	0.754	0.728	0.728
PROMISE-Lucene-2.2	0.714	0.714	0.714
PROMISE-Tomcat	0.758	0.805	0.805
PROMISE-Xalan-2.4	0.731	0.720	0.720
PROMISE-Xalan-2.5	0.663	0.656	0.666
PROMISE-Xalan-2.6	0.798	0.983	0.797
AEEEM-EQ	0.836	0.841	0.828
AEEEM-JDT	0.780	0.820	0.784
AEEEM-LC	0.868	0.803	0.764
AEEEM-ML	0.787	0.805	0.804
AEEEM-PDE	0.737	0.697	0.748
NASA-Cm1	0.519	0.656	0.588
NASA-Mw1	0.613	0.561	0.535
NASA-PC1	0.900	0.895	0.907
NASA-PC3	0.810	0.798	0.801
NASA-PC4	0.911	0.926	0.931
RELINK-Apache	0.683	0.789	0.684
RELINK-Safe	0.857	0.857	0.700
RELINK-Zxing	0.738	0.747	0.717
Average (Avg)	**0.749**	**0.767**	**0.741**

datasets, its performance was comparable to either the GWO-based model or the full dataset.

Table 6.5 presents the computed ROC-AUC value by AB for the entire datasets and the processed datasets, where the features were selected by GWO and 2M-GWO. It is observed that the 2M-GWO-based model outperforms the GWO and WFS case in terms of overall performance, with an average ROC-AUC value improvement of 4.6%. Specifically, for approximately 33% of the datasets, the 2M-GWO-based model performs better, while for around 11% of the datasets, it performs similarly to either of the two approaches.

According to the data presented in Table 7.3 , the 2M-GWO-KNN demonstrates superior classification performance compared to the other model in over 50% of the

Table 6.5: ROC-AUC Scores by GB

Projects	**GWO**	**2M-GWO**	**WFS**
PROMISE-Ant-1.7	0.741	0.793	0.693
PROMISE-Camel-1.2	0.507	0.517	0.517
PROMISE-Camel-1.4	0.732	0.714	0.714
PROMISE-Jedit-3.2	0.791	0.780	0.780
PROMISE-Jedit-4.1	0.789	0.822	0.714
PROMISE-Jedit-4.2	0.673	0.692	0.714
PROMISE-Log4j-1.0	0.757	0.793	0.657
PROMISE-Log4j-1.1	0.752	0.933	0.933
PROMISE-Lucene-2.0	0.770	0.767	0.767
PROMISE-Lucene-2.2	0.573	0.704	0.704
PROMISE-Tomcat	0.659	0.731	0.721
PROMISE-Xalan-2.4	0.738	0.734	0.724
PROMISE-Xalan-2.5	0.680	0.806	0.648
PROMISE-Xalan-2.6	0.807	1.000	0.806
AEEEM-EQ	0.766	0.845	0.818
AEEEM-JDT	0.819	0.825	0.831
AEEEM-LC	0.845	0.778	0.577
AEEEM-ML	0.787	0.786	0.803
AEEEM-PDE	0.673	0.643	0.702
NASA-Cm1	0.588	0.603	0.642
NASA-Mw1	0.570	0.600	0.613
NASA-PC1	0.909	0.899	0.916
NASA-PC3	0.796	0.768	0.705
NASA-PC4	0.910	0.920	0.924
RELINK-Apache	0.732	0.775	0.589
RELINK-Safe	0.914	0.914	0.729
RELINK-Zxing	0.601	0.715	0.719
Average (Avg)	**0.736**	**0.772**	**0.728**

datasets, as indicated by the ROC-AUC value. The average ROC-AUC value for the overall performance of 2M-GWO shows an increase of approximately 4% when compared to the other two models.

Based on the analysis presented in Table 6.7 and Figure 6.5, on taking an average value, the 2M-GWO algorithm achieved an approximate 80% reduction in feature sets across various datasets. The NASA dataset showed the most significant reduction at around 86%, followed by the AEEEM and RELINK datasets, which saw an 80% reduction each. In contrast, the PROMISE dataset had the smallest reduction in features at 75

Table 6.6: ROC-AUC Scores by KNN

Projects	**GWO**	**2M-GWO**	**WFS**
PROMISE-Ant-1.7	0.741	0.793	0.693
PROMISE-Camel-1.2	0.507	0.517	0.517
PROMISE-Camel-1.4	0.732	0.714	0.714
PROMISE-Jedit-3.2	0.791	0.780	0.780
PROMISE-Jedit-4.1	0.789	0.822	0.714
PROMISE-Jedit-4.2	0.673	0.692	0.714
PROMISE-Log4j-1.0	0.757	0.793	0.657
PROMISE-Log4j-1.1	0.752	0.933	0.933
PROMISE-Lucene-2.0	0.770	0.767	0.767
PROMISE-Lucene-2.2	0.573	0.704	0.704
PROMISE-Tomcat	0.659	0.731	0.721
PROMISE-Xalan-2.4	0.738	0.734	0.724
PROMISE-Xalan-2.5	0.680	0.806	0.648
PROMISE-Xalan-2.6	0.807	1.000	0.806
AEEEM-EQ	0.766	0.845	0.818
AEEEM-JDT	0.819	0.825	0.831
AEEEM-LC	0.845	0.778	0.577
AEEEM-ML	0.787	0.786	0.803
AEEEM-PDE	0.673	0.643	0.702
NASA-Cm1	0.588	0.603	0.642
NASA-Mw1	0.570	0.600	0.613
NASA-PC1	0.909	0.899	0.916
NASA-PC3	0.796	0.768	0.705
NASA-PC4	0.910	0.920	0.924
RELINK-Apache	0.732	0.775	0.589
RELINK-Safe	0.914	0.914	0.729
RELINK-Zxing	0.601	0.715	0.719
Average (Avg)	**0.736**	**0.772**	**0.728**

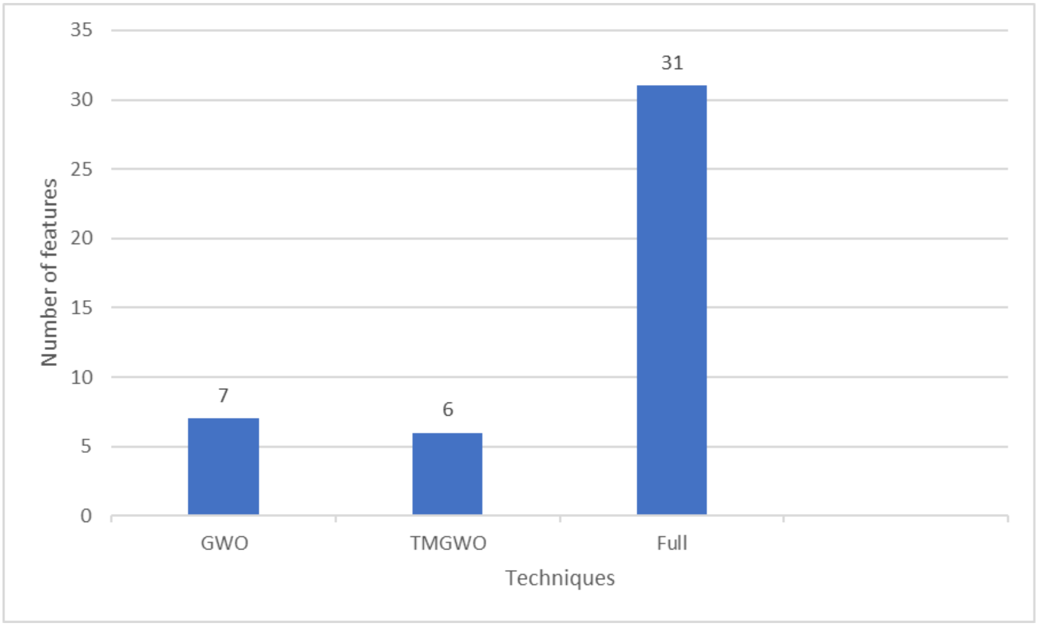


Figure 6.5: Comparison of GWO, 2M-GWO, and WFS techniques on the basis of the number of features selected

Table 6.7: Number of Features Calculated by GWO, 2M-GWO, and WFS Across Various Projects

Projects	**GWO**	**2M-GWO**	**WFS**
PROMISE-Ant-1.7	5	6	20
PROMISE-Camel-1.2	6	6	20
PROMISE-Camel-1.4	5	6	20
PROMISE-Jedit-3.2	5	5	20
PROMISE-Jedit-4.1	5	4	20
PROMISE-Jedit-4.2	4	4	20
PROMISE-Log4j-1.0	4	4	20
PROMISE-Log4j-1.1	3	3	20
PROMISE-Lucene-2.0	6	5	20
PROMISE-Lucene-2.2	7	6	20
PROMISE-Tomcat	4	4	20
PROMISE-Xalan-2.4	6	5	20
PROMISE-Xalan-2.5	10	7	20
PROMISE-Xalan-2.6	7	6	20
AEEEM-EQ	11	12	61
AEEEM-JDT	14	13	61
AEEEM-LC	6	6	61
AEEEM-ML	12	14	61
AEEEM-PDE	12	13	61
NASA-Cm1	4	2	37
NASA-Mw1	4	4	37
NASA-PC1	4	4	37
NASA-PC3	8	6	37
NASA-PC4	8	8	37
RELINK-Apache	5	6	26
RELINK-Safe	3	3	26
RELINK-Zxing	8	7	26
Average (Avg)	**7**	**6**	**31**

Table 6.8: Mean Rank Obtained on the Basis of ROC-AUC for Feature Selection Techniques

Techniques	**Mean Rank (p-value=0.022)**
2M-GWO	1.00 (I)
GWO	2.40 (II)
WFS	2.60 (III)

Table 6.8 presents the comparative performance of different feature selection techniques based on their mean rank achieved by Friedman's test, determined through the ROC-AUC metric. The mean rank serves as a measure of each technique's efficacy in enhancing the prediction accuracy of a software defect prediction model. A lower mean rank indicates better performance.

The p-value associated with these rankings is 0.022, suggesting that the differences in performance among these techniques are statistically significant at the 95% confidence interval. 2M-GWO stands out as the most effective technique, achieving the highest performance with a mean rank of 1.00, denoted as rank I. GWO is ranked second with a mean rank of 2.40, labelled as rank II. This shows that the standard Grey Wolf Optimiser, while effective, is outperformed by its two-phase variant in the context of feature selection for defect prediction. WFS is closely ranked third with a mean rank of 2.60, marked as rank III.

6.3.2 Result Analysis based on RQ2

RQ 2 Which is the best-performing ML technique for predicting software defects?

Table 6.9: Mean Rank Obtained on the Basis of ROC-AUC for ML Techniques

Classification Techniques	**Mean Rank (p-value=0.034)**
2M-GWO-RF	2.19 (I)
2M-GWO-KNN	2.94 (II)
2M-GWO-SVM	3.15 (III)
2M-GWO-AB	3.35 (IV)
2M-GWO-GB	3.37 (V)

Table 6.9 displays the mean ranks obtained by the Friedman's test on the basis of ROC-AUC for the ML techniques used in the study. 2M-GWO-RF achieved the best mean rank of 2.19, indicating its superior performance in terms of prediction accuracy and efficiency.

2M-GWO-KNN attained a mean rank of 2.94, placing it second in terms of performance. While it is not as effective as 2M-GWO-RF, it still demonstrates good performance. With a mean rank of 3.15, 2M-GWO-SVM is ranked third in terms of performance.

AB and GB are the ensemble classifiers with mean ranks of 3.35 and 3.37, ranking fourth and last among the classifiers evaluated. The p-value of 0.034 indicates that the differences in the mean ranks of these techniques are statistically significant.

Overall, the combination of 2M-GWO and Random Forest (2M-GWO-RF) appears to offer the best performance among the tested classification techniques.

6.3.3 Result Analysis based on RQ3

RQ 3. Which software metrics are frequently selected using the proposed algorithm?

The analysis of feature selection across various datasets using the 2M-GWO FS technique is detailed in Figures 4 to 7. These figures graphically represent the selection frequency of specific software metrics (features) crucial for predicting software defects. The x-axis of each figure indicates different software metrics derived from the datasets, while the y-axis quantifies how frequently each metric was selected across multiple iterations.

For the RELINK dataset (Figure 6.6), metrics such as CountLineCodeDecl, CountLineBlank, and CountLineCode show high selection frequencies of 26, 21, and 20 times, respectively, out of 60 total runs distributed across three projects (each executed 20 times). In the case of the PROMISE dataset (Figure 6.7), the object-oriented metrics rfc, ce, and npm were selected 131 and 102 times each out of 280 iterations. Regarding the AEEEM dataset (Figure 6.8), which includes 61 process metrics across five projects, metrics such as CVSEntropy, numberOfNontrivialBugsFoundUntil, and Ck OO numberOfMethodsInherited were most frequently selected, with counts of 49, 45, and 41 respectively, out of 100 runs. For the NASA project (Figure 6.9), which focuses on Halstead's and size metrics, the metrics LOC CODE AND COMMENT, LOC BLANK, and LOC EXECUTABLE were identified as the most frequently occurring, with each being selected 28, 25, and 25 times respectively in twenty runs across five projects.

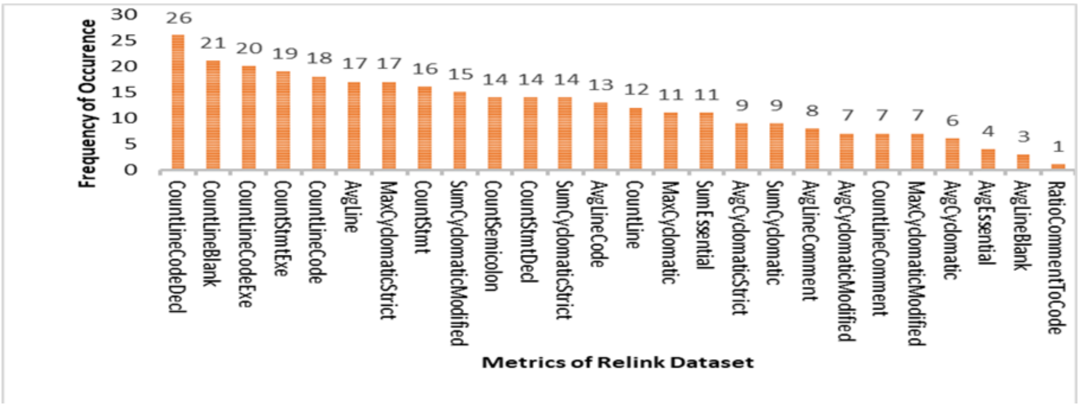


Figure 6.6: Occurrence of metrics for RELINK Dataset

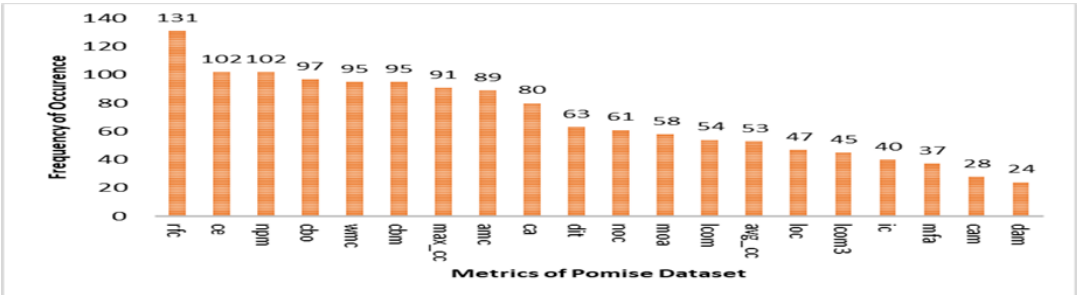


Figure 6.7: Occurrence of metrics for PROMISE Dataset

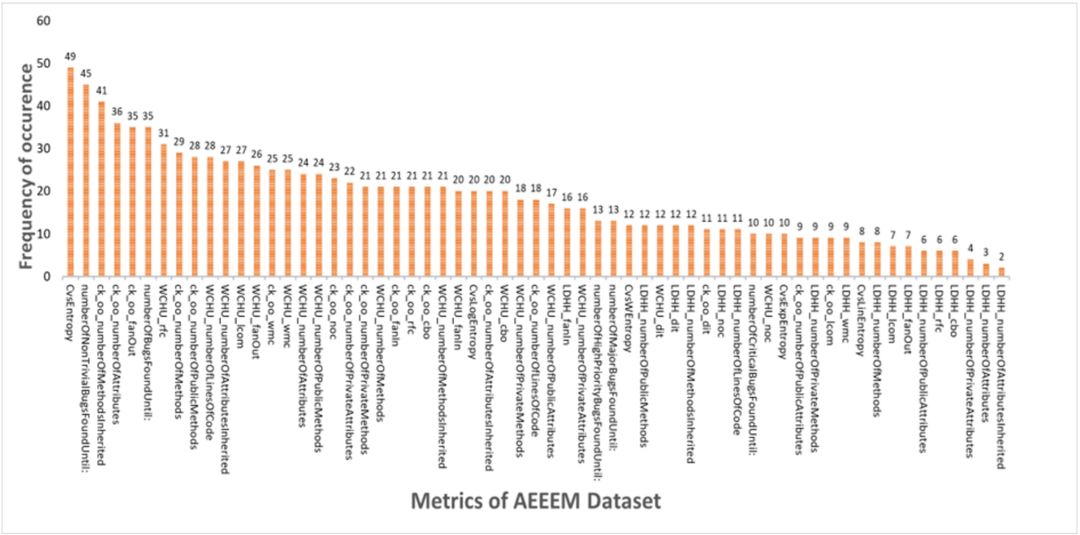


Figure 6.8: Occurrence of metrics for AEEEM Dataset

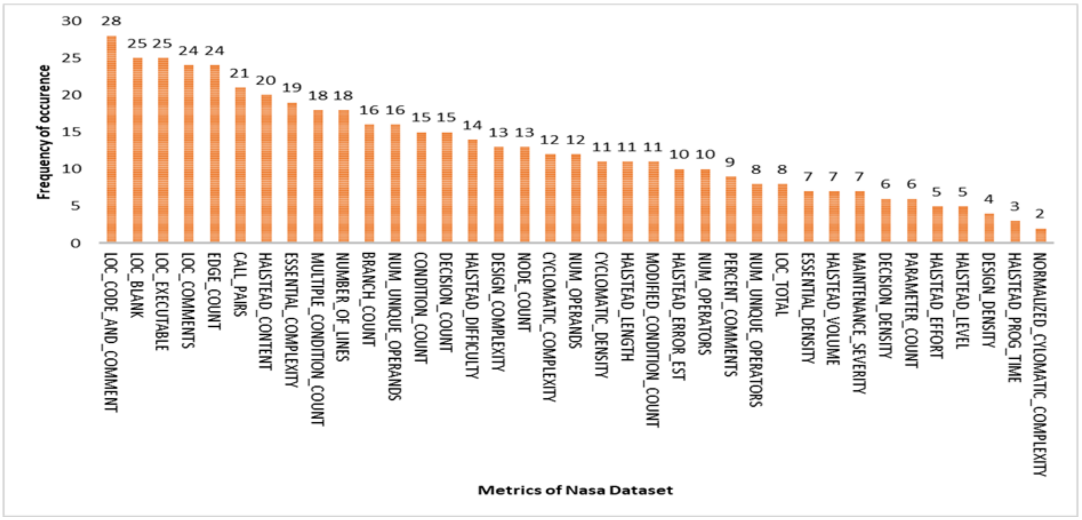


Figure 6.9: Occurrence of metrics for NASA Dataset

6.3.4 Result Analysis based on RQ3

RQ 4 What is the predictive performance of the proposed model as compared to other benchmark FS models for SDP

In order to conduct a comprehensive comparative analysis, two types of analysis are made.

In the first type, the proposed algorithm is evaluated against five closely related meta-heuristic algorithms: Harris Hawks Optimization (HHO) [169], Salp Swarm Optimization (SSO) [170], Whale Optimization (WO) [171], Jaya Optimization (JO) [172], and Sine Cosine Optimization (SCO) [173]. These comparisons, summarized in Table 6.10, were carried out under consistent experimental conditions to ensure fairness and accuracy in the results. Each algorithm was implemented and executed within the same computational environment, adhering to identical parameter settings.

The ROC-AUC measure was used as the primary basis for evaluating performance across all algorithms, as it provides a comprehensive measure of an algorithm's ability to distinguish between classes or optimize solutions effectively. By examining the

AUC scores, we analyzed the fact that the proposed technique outperforms the other techniques in approximately 55.5% of the total datasets. SSO obtained a promising outcome in approximately 18.5% of the datasets. HHO and WO resulted in the best ROC_AUC value for two datasets, whereas SCO and JO outperformed for only one dataset each. This showcases the effectiveness of the proposed method in selecting the most pertinent features, which notably boost the performance of the SDP model.

Table 6.10: ROC_AUC Values for 2M-GWO-RF and Other Metaheuristic Techniques

Projects	**2M-GWO-RF**	**SCO-RF**	**SSO-RF**	**JO-RF**	**WO-RF**	**HHO-RF**
PROMISE-Ant-1.7	0.845	0.824	0.830	0.829	0.820	0.818
PROMISE-Camel-1.2	0.634	0.625	0.604	0.615	0.630	0.615
PROMISE-Camel-1.4	0.710	0.787	0.729	0.753	0.765	0.798
PROMISE-Jedit-3.2	0.776	0.799	0.799	0.808	0.784	0.831
PROMISE-Jedit-4.1	0.862	0.807	0.842	0.813	0.809	0.850
PROMISE-Jedit-4.2	0.730	0.650	0.615	0.685	0.712	0.727
PROMISE-Log4j-1.0	0.889	0.875	0.817	0.865	0.812	0.801
PROMISE-Log4j-1.1	1.000	0.954	1.000	0.957	0.964	0.997
PROMISE-Lucene-2.0	0.767	0.767	0.760	0.769	0.789	0.785
PROMISE-Lucene-2.2	0.616	0.645	0.664	0.614	0.623	0.613
PROMISE-Tomcat	0.830	0.799	0.802	0.772	0.825	0.809
PROMISE-Xalan 2.4	0.749	0.804	0.849	0.816	0.718	0.788
PROMISE-Xalan 2.5	0.827	0.824	0.824	0.823	0.877	0.812
PROMISE-Xalan-2.6	1.000	0.988	0.998	0.991	0.962	0.990
AEEEM-EQ	0.841	0.897	0.839	0.896	0.875	0.807
AEEEM-JDT	0.881	0.814	0.881	0.805	0.803	0.817
AEEEM-LC	0.821	0.820	0.803	0.805	0.803	0.805
AEEEM-ML	0.838	0.822	0.837	0.801	0.811	0.808
AEEEM-PDE	0.784	0.775	0.709	0.773	0.823	0.807
NASA-Cm1	0.718	0.708	0.682	0.705	0.713	0.674
NASA-Mw1	0.500	0.655	0.604	0.615	0.680	0.691
NASA-PC1	0.925	0.888	0.894	0.905	0.893	0.817
NASA-PC3	0.835	0.870	0.886	0.884	0.872	0.818
NASA-PC4	0.931	0.917	0.902	0.879	0.892	0.852
RELINK-Apache	0.697	0.655	0.604	0.615	0.618	0.691
RELINK-Safe	0.786	0.737	0.817	0.785	0.802	0.816
RELINK-Zxing	0.827	0.773	0.813	0.886	0.836	0.814

In the second type, the performance outcomes of established approaches are sourced from existing literature and subjected to a thorough comparison. In the RKEE-based model, Riaz et al. [174] proposed two-stage data pre-processing which incorporates FS and a new rough set Easy Ensemble scheme, which shows an average value of 0.771, whereas the existing model proposed by Tumar et al. [70] was based on improved binary Moth Flame Optimization with Adaptive synthetic sampling

(ADASYN) to predict software defects. BMFO is employed as a wrapper FS, while ADASYN enhances the input dataset, addresses the imbalanced dataset and shows a ROC-AUC value of 0. 7552. When compared with the existing methods based on evolutionary FS techniques, the proposed model based on 2M-GWO and Random Forest classifier obtained a ROC-AUC of 0.800, which is better than the existing methods.

6.4 Discussion

In this chapter, we proposed a novel prediction model based on the Two-Phase Grey Wolf Optimizer (2M-GWO) for Software Defect Prediction (SDP). This model uniquely integrates feature selection (FS) to improve the predictive performance of various machine learning (ML) classifiers, specifically Support Vector Machine (SVM), AdaBoost (AB), Gradient Boosting (GB), NaÃ¯ve Bayes (NB), and k-Nearest Neighbors (KNN). These classifiers were employed to evaluate the binary classification task on a dataset constructed from 27 well-established software defect datasets, namely AEEEM, PROMISE, RELINK, and NASA datasets.

The experiments were systematically designed, and the results demonstrated that the 2M-GWO method provides significant benefits for software defect prediction. The primary findings of this study are outlined as follows:

1. The two-phase Grey Wolf Optimizer-based feature selection, in conjunction with the Random Forest (RF) classifierâtermed 2M-GWO-RFâproved highly effective in SDP tasks. The FS method played a pivotal role in reducing the feature space, which in turn enhanced model performance across different datasets. By eliminating redundant and irrelevant features, the model could focus on the most impactful metrics, leading to more accurate predictions and

improved computational efficiency.

2. The 2M-GWO-RF model showed superior performance compared to other models, particularly in terms of the Receiver Operating Characteristic - Area Under the Curve (ROC_AUC) metric. With a mean rank of 2.19 for ROC_AUC, this model demonstrated remarkable prediction accuracy and efficiency, answering Research Question 2 (RQ2). This result underscores the modelâs ability to maintain high predictive performance across different data sources, further validating the robustness of the proposed approach.

3. To statistically validate the performance of the 2M-GWO-RF model, we employed Friedmanâs test to compare mean rank differences among the models. The results showed a significant difference ($p = 0.034$), reinforcing the efficacy of the 2M-GWO-RF model as a top classification technique within the examined classifiers. This test further supports the hypothesis that 2M-GWO-RFâs feature selection approach contributes meaningfully to its predictive success.

4. The 2M-GWO-RF modelâs robustness is evident through its performance on diverse datasets. Important metricsâsuch as CountLineCodeDecl, RFC (Response for Class), CE (Coupling between Classes), NPM (Number of Public Methods), CVSEntropy, and LOC CODE AND COMMENTâwere consistently identified as influential features, highlighting their relevance in defect prediction. The consistent effectiveness across these metrics indicates the modelâs adaptability to varied datasets and suggests that these features may be crucial for future data collection and SDP model development.

5. Compared to five other metaheuristic techniques, the 2M-GWO-RF model outperformed in more than 55% of the datasets, positioning it as a superior choice for software defect prediction. This result underlines the modelâs advanced ca-

pability to leverage feature selection more effectively than other state-of-the-art methods, making it a promising alternative for practitioners seeking improved prediction performance and model interpretability.

In conclusion, the two-phase Grey Wolf Optimizer with Random Forest classifier 2M-GWO-RF shows significant promise for enhancing software defect prediction. By providing a powerful feature selection mechanism and achieving high prediction accuracy across multiple datasets, this model offers a robust tool for defect prediction tasks. The findings suggest that the 2M-GWO-RF approach not only enhances predictive capability but also sets a foundation for further advancements in feature selection and optimization techniques in software engineering.

Chapter 7

Impact of Hyperparameter tuning on Software Defect Prediction Model

7.1 Introduction

In recent years, machine learning (ML) and deep learning (DL) techniques have been widely adopted in Software Defect Prediction (SDP) [40]. Algorithms such as Naive Bayes, random forests (RF), support vector machines (SVMs), and logistic regression (LR), as well as DL models like Convolutional Neural Networks (CNNs), Long Short-Term Memory networks (LSTMs), and Gated Recurrent Units (GRUs), have demonstrated effectiveness in identifying complex patterns in software [175, 176]. However, the performance of these models is highly influenced by their hyperparameters, which dictate learning rates, the number of neurons, batch sizes, and other key factors in the training process. Optimizing these hyperparameters is essential to achieving accurate predictions, avoiding issues like underfitting or overfitting, and ensuring that models generalize well to unseen data.

Hyperparameter tuning (HPT) plays a critical role in enhancing the performance of SDP models. Standard models with default hyperparameters often underperform in SDP tasks, especially when dealing with imbalanced datasets, which are common in defect prediction. For example, tuning hyperparameters in regression models has been shown to improve prediction accuracy by over 15%, while optimally tuned DL models can see performance boosts of up to 40%. HPT frameworks like Optuna facilitate the systematic search for optimal hyperparameters, making it easier to identify the best configurations for both traditional ML and DL models [177].

Furthermore, HPT can also optimize preprocessing techniques such as the Synthetic Minority Over-sampling Technique (SMOTE), which addresses class imbalance issues by augmenting minority class samples. Studies have shown that optimized versions of SMOTE, such as SMOTUNED, can enhance the performance of neural networks and other classifiers on imbalanced defect datasets. Through frameworks like Bayesian optimization (BO) and differential evolution (DE), HPT has proven effective in fine-tuning both classifiers and sampling techniques, yielding more robust SDP models [178–182].

This chapter aims to examine the impact of HPT on SDP models, particularly by employing the Optuna framework to optimize DL models and ensemble approaches as well as preprocessing techniques. By leveraging optimized hyperparameters, the study demonstrates substantial improvements in prediction accuracy, model stability, and computational efficiency. This chapter synthesizes insights from two novel studies focused on the role of HPT in enhancing SDP models. The first study introduces a tailored approach for balancing imbalanced datasets, which is a common challenge in defect prediction. The second study explores HPT techniques for DL models in SDP, proposing a robust ensemble-based solution. Together, these studies demonstrate how HPT can elevate the performance of defect prediction models, showcasing improvements across various predictive metrics.

7.2 Research Methodology

This chapter presents two innovative methods OpTunedSMOTE and a Stacked Ensemble Approach or hyperparameter tuning in Software Defect Prediction (SDP) using the Optuna framework. The OpTunedSMOTE model optimizes only SMOTE parameters to effectively handle class imbalance, improving classifier performance for imbalanced datasets. Meanwhile, the Stacked Ensemble Approach combines CNN, LSTM, and GRU as base learners with Random Forest (RF) as the meta-learner, optimizing both SMOTE and classifier parameters. These techniques leverage Optuna's optimization to achieve higher prediction accuracy and model robustness. Each approach is structured to address unique challenges in SDP, as illustrated in the comprehensive workflows provided.

7.2.1 Research Methodology of Study 1

The proposed approach leverages the Optuna [183] framework to perform efficient and effective hyperparameter tuning, ensuring that the classifiers operate at their optimal capacity. By integrating advanced optimization algorithms with robust evaluation metrics, our methodology aims to provide a scalable and generalizable solution for improving the accuracy and reliability of classification models across various datasets.

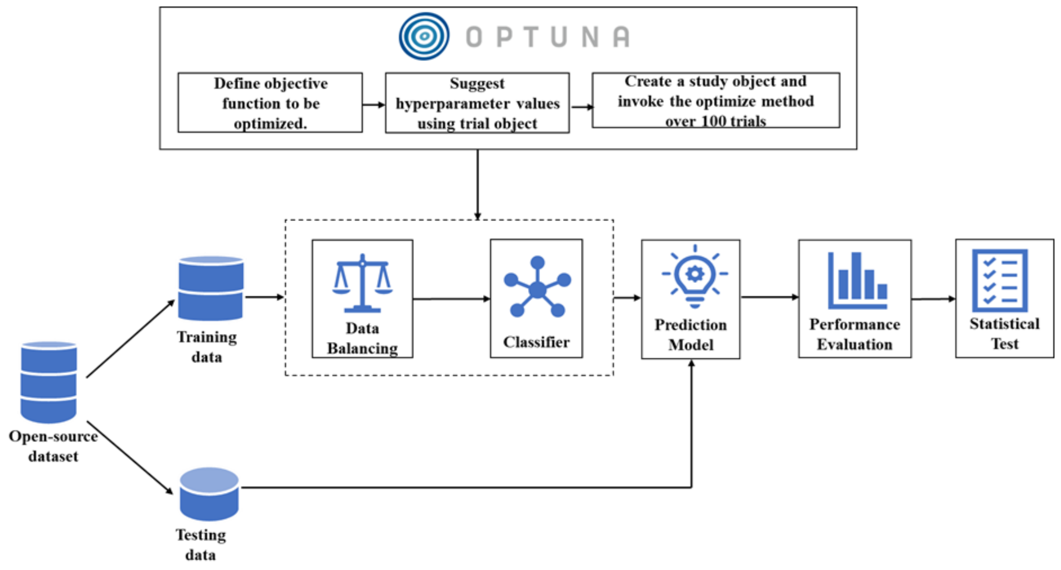

OPTUNA
Define objective function to be optimized.
Suggest hyperparameter values using trial object
Create a study object and invoke the optimize method over 100 trials
Training data
Data Balancing
Classifier
Prediction Model
Performance Evaluation
Statistical Test
Open-source dataset
Testing data

Figure 7.1: A detailed framework for the OpTunedSMOTE approach

Figure 7.1 illustrates a comprehensive workflow for optimizing classification models using the Optuna framework. Data is taken from open-source repositories. Then, the data is further divided into training and test data. Before training the classifier, data balancing techniques are applied to training data to address class imbalance issues. This step ensures that the model is not biased towards the majority class. The balanced data is fed into a classifier. In the case of the OpTunedSMOTE approach, only parameters of the SMOTE technique are tuned. In the case of BOTH, the hyperparameters of both SMOTE and classifiers are optimized using the Optuna framework. The BOTH approach is included to compare the effectiveness of the OpTunedSMOTE approach. The classifier generates a prediction model based on the training data. The performance of the prediction model is evaluated using appropriate performance metrics. This step ensures that the model meets the desired criteria. Finally, statistical tests are conducted to validate the model's performance and robustness.In the HPT of parameters using Optuna, the first step involves defining the objective function that needs to be optimized. This function encapsulates the model training and evaluation process, and it is the metric we aim to improve. Optuna uses a trial object to suggest different hyperparameter values. These suggestions are generated based on the objective function. A study object is created, and the optimized method is invoked to perform the optimization over 100 trials. Each trial evaluates a set of hyperparameters and records the performance. The workflow ensures a systematic approach to optimize and evaluate classification models, leveraging the capabilities of Optuna for HPT and employing a robust framework for model development and assessment.

7.2.2 Research Methodology of Study 2

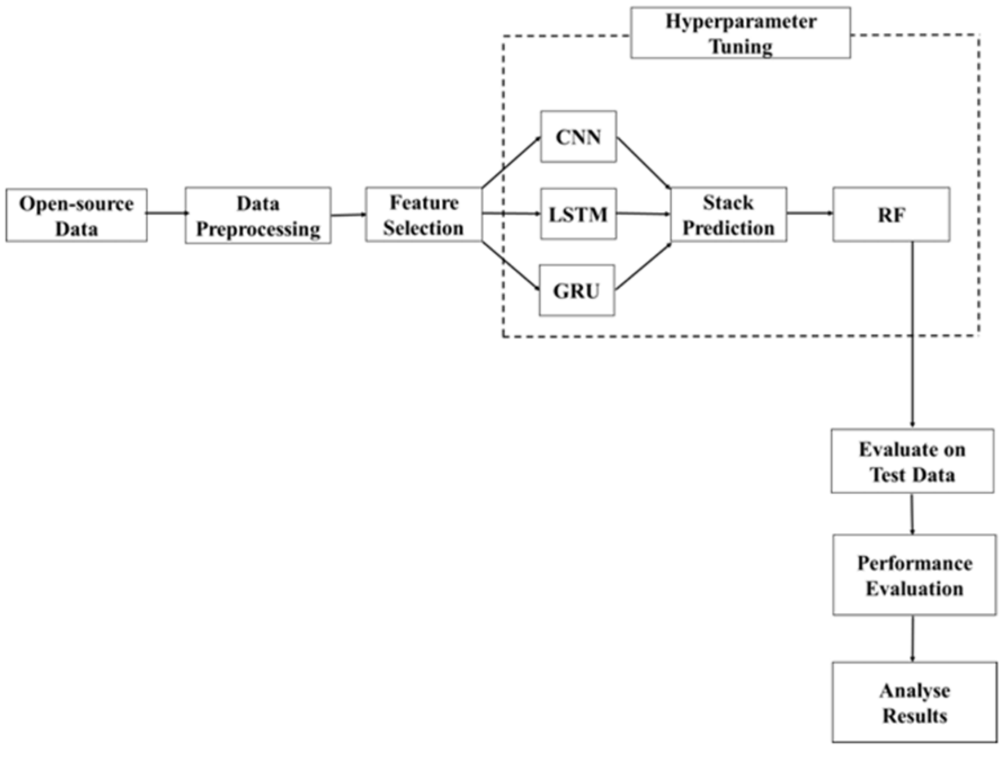


Figure 7.2: A detailed framework for the Stacked Ensemble approach

In this part, we introduce our proposed model which utilizes an ensemble of CNN [99, 184], GRU [185, 186], and LSTM with RF as the meta learner. Additionally, we include hyperparameter tuning (HPT) using Optuna. We have obtained the datasets from the PROMISE repository. In addition, we have used Recursive Feature Elimination (RFE) as a strategy for selecting the most relevant characteristics for the target class. The datasets are divided into training and testing datasets to train and evaluate the proposed models. Eventually, we developed and assessed our models using the ROC-AUC metric. Figure 7.2 depicts the whole workflow of the proposed Software Defect Prediction (SDP) technique.

7.3 Experimental Framework

In this section, we outline the experimental framework designed to implement and evaluate both the OpTunedSMOTE and Stacked Ensemble approaches for SDP. Each study follows a structured process encompassing data preprocessing, hyperparameter tuning, model training, and performance evaluation. For Study 1 (OpTunedSMOTE), we focus on optimizing SMOTE parameters alone to address class imbalance effectively, while in Study 2 (Stacked Ensemble), we employ a combination of CNN, LSTM, and GRU as base learners, with RF as the meta-learner, optimizing both SMOTE and classifier hyperparameters. This framework ensures a systematic approach to assessing each model's accuracy and robustness across diverse datasets.

7.3.1 Dataset Collection

For Study 1, three criteria guided the data selection process:

- **C1: Accessibility of Public Defect Datasets:** To reduce bias and ensure reproducibility, we used publicly accessible datasets from varied corpora and domains, acknowledging the impact dataset choice has on model performance.

- **C2: Minimum Events Per Variable (EPV) Threshold:** Following recommendations, we included datasets with EPV values above 10 to avoid overfitting risks associated with small sample sizes.

- **C3: Defect Ratio Threshold:** Datasets with a defect ratio under 50% were selected to prevent model bias toward the majority class.

The study analyzed 17 datasets: Camel-1.2, Debug, Derby-10.2.1.6, Derby-10.3.1.4, JDT, JM1, ML, PC5, PDE, Prop-1 to Prop-5, SWT, Xalan-2.5, and Xalan-2.6.

For Study 2, we selected six open-source Java projects, namely Ant-1.7, Camel-1.6, Ivy-2.0, Jedit-4.3, Log4j-1.2, Xerces-1.4, from the PROMISE dataset, accessible at `https://www.kaggle.com/datasets/nazgolnikravesh/software\protect\penalty\z@-defect-prediction-dataset`. These datasets represent diverse project scales, with instance counts from 205 to 965 and defect rates ranging from 2.23% to 92.19%.

7.3.2 Data Preprocessing

In both studies, data preprocessing is a critical stage aimed at enhancing the accuracy and robustness of the ML models developed for SDP. Effective preprocessing involves addressing key issues such as data imbalance, feature selection, and normalization, which impact the overall quality and performance of predictive models.

In Study 1, the Synthetic Minority Over-sampling Technique (SMOTE) is employed to resolve class imbalance, a common issue where one class significantly outnumbers the other. Class imbalance can bias models toward the majority class, leading to poor performance on minority class predictions. SMOTE counters this by generating synthetic samples of the minority class, which improves model generalization and reduces bias. Building on this, an advanced version called OpTunedSMOTE is introduced, which leverages the Optuna framework for hyperparameter tuning to customize SMOTE parameters according to specific dataset characteristics, aiming to achieve consistent results across various datasets and models.

Study 2 applies Min-Max normalization to scale data within a specific range, making optimization processes more effective and reducing the influence of outliers. Feature selection is another key step, where Recursive Feature Elimination (RFE) is utilized to retain only the most significant features. By iteratively removing less impactful features, RFE enhances the model's efficiency, focusing on critical predictors and thus improving SDP model accuracy and interpretability.

7.3.3 Hyperparameter Tuning Setup

Hyperparameter tuning (HPT) is essential in Software Defect Prediction (SDP) because it significantly impacts the performance of machine learning (ML) models. Default hyperparameters often lead to suboptimal results, and tuning them can improve the predictive power of models by up to 40% [4,5]. Several techniques are employed for HPT in SDP. Grid Search method involves systematically searching through all possible combinations of hyperparameters to find the optimal set. While it is exhaustive, it can be computationally expensive. Random Search method involves randomly sampling hyperparameters from a specified range. Although it is less computationally expensive than grid search but has several disadvantages. It lacks systematic exploration, potentially missing optimal combinations. Its effectiveness depends heavily on the sampling strategy and number of iterations. Bayesian Optimization uses Bayesian inference to iteratively update the hyperparameter settings based on previous results. It uses an informed search strategy, creating a probabilistic model of the objective function and updating it iteratively. By learning from previous evaluations, it directs the search towards more promising regions, effectively balancing exploration and exploitation. This method also excels in handling continuous hyperparameters.

In both studies, Optuna is employed for hyperparameter tuning (HPT) to optimize model performance using Bayesian optimization principles. Optuna is a sophisticated and flexible hyperparameter optimization framework designed to automate and accelerate the optimization process for machine learning models, offering a unified interface within the Python programming language. OPTUNA has gained popularity for hyperparameter optimization due to its distinct advantages [187]. These advantages are outlined as follows:

- Define-by-Run Style API: This flexible approach allows users to dynamically establish the hyperparameter search space, facilitating efficient and targeted optimization [87].
- Ease of Setup: Optuna's architecture is versatile and user-friendly, enabling smooth configuration for both lightweight experiments and large-scale distributed computations [185].
- Pruning and Sampling Mechanism: Optuna incorporates various sampling algorithms, such as GridSearch, RandomSearch, TPE, and CMA-ES. Pruning helps conserve computational resources by discontinuing unpromising trials early in the process [136, 188, 189].

Figure 7.3 illustrates the architectural design of the Optuna framework. In the Optuna hyperparameter optimization process, each study involves executing an objective function by each worker. The objective function utilizes Optuna APIs to conduct trials. By leveraging the API, the objective function can access shared storage and retrieve historical study data when required. Each worker runs the objective function independently and communicates progress on the study through the shared storage facility.

[h]

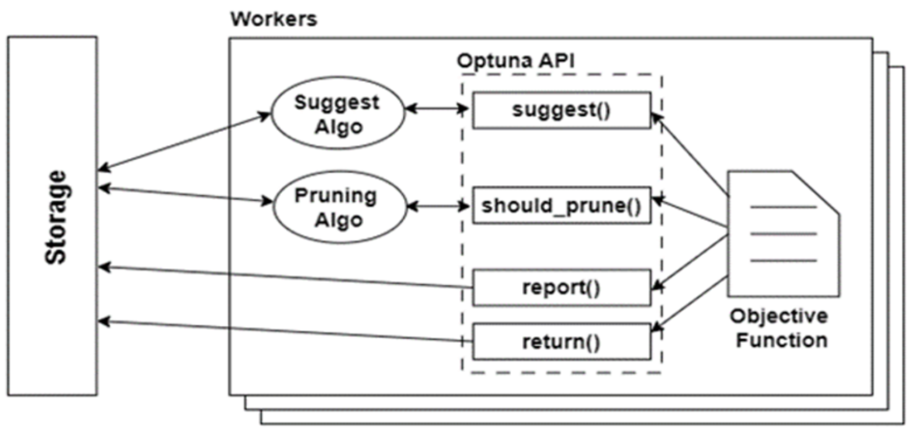


Figure 7.3: Architectural design of the Optuna framework.

The parameter details of the tuned parameter for study 1 is mentioned in Table 7.1 and for study 2 it is in Table 7.2.

7.3.4 Model Training and Testing Framework

In this chapter, the 10-fold cross-validation method is employed for both studies to ensure the robustness and generalizability of the models. The significance of using 10-fold cross-validation lies in its ability to maximize the use of available data while providing a reliable measure of model performance. The in-depth description of this method, is present in Section 2.7 in Chapter 2, where it is defined and discussed in detail.

Table 7.1: Parameter descriptions for techniques used in study 1.

Technique	Parameter	Description	Datatype	Range
SMOTE	sampling_strategy	Sampling information to resample the data set.	str	{'minority', 'not minority', 'not majority', 'all'}
	k_neighbors	The nearest neighbours are used to define the neighbourhood of samples to use to generate the synthetic samples.	int	[1, 20]
RF	n_estimators	The number of trees in the forest.	int	[100, 1000]
	max_depth	The maximum depth of the tree.	int	[2, 20]
	min_samples_split	The minimum number of samples required to split an internal node.	int	[2, 10]
	min_samples_leaf	The minimum number of samples required to be at a leaf node.	int	[1, 10]
SVM	C	Regularization parameter.	float	[0.1, 10]
	kernel	Specifies the kernel type to be used in the algorithm.	str	{'linear', 'poly', 'rbf', 'sigmoid'}
MLP	alpha	Strength of the L2 regularization term.	float	[0.00001, 0.001]
	activation	Activation function for the hidden layer.	str	{'relu', 'tanh', 'logistic'}
KNN	n_neighbors	Number of neighbours.	int	[3, 50]
	weights	Weight function used in prediction.	str	{'uniform', 'distance'}
	p	Power parameter for the Minkowski metric.	float	[1, 2]
XGB	n_estimators	The number of boosting rounds.	int	[100, 1000]
	max_depth	Maximum tree depth for base learners.	int	[3, 10]
	learning_rate	Boosting learning rate.	float	[0.01, 0.1]
	subsample	Subsample ratio of the training instance.	float	[0.5, 1]
	gamma	Minimum loss reduction required to make a further partition on a leaf node of the tree.	float	[0, 5]

7.3.5 Evaluation Metrics

In this chapter, we assess the performance of Software Defect Prediction (SDP) models using ROC-AUC and MCC performance metrics to provide a comprehensive evaluation of the proposed techniques. For Study 1, the primary metrics used include Receiver Operating Characteristic Area Under the Curve (ROC-AUC) and Matthew's Correlation Coefficient (MCC), both of which capture model accuracy and the balance between false positives and negatives. Additionally, we measure execution time

Table 7.2: Parameter descriptions for techniques used in study 2.

Technique	Parameter Name	Description	Datatype	Range
GRU	gru_units1	Number of units in the first GRU layer.	int	[16, 64]
	gru_units2	Number of units in the second GRU layer.	int	[32, 128]
CNN	conv_filters1	Number of filters in the first convolutional layer.	int	[16, 64]
	conv_filters2	Number of filters in the second convolutional layer.	int	[32, 128]
	conv_kernel_size1	Size of the kernel in the first convolutional layer.	int	[3, 5]
	conv_kernel_size2	Size of the kernel in the second convolutional layer.	int	[3, 5]
LSTM	lstm_units1	Number of units in the first LSTM layer.	int	[16, 64]
	lstm_units2	Number of units in the second LSTM layer.	int	[32, 128]
Common (GRU, CNN, LSTM)	dense_units	Number of units in the dense (fully connected) layer.	int	[32, 128]
	learning_rate	Learning rate for model optimization.	float	[0.0001, 0.01]
	batch_size	Number of samples per gradient update.	categorical	[32, 128]
	activation	Activation function used in the layers.	categorical	{'relu', 'tanh', 'sigmoid'}
	optimizer	Optimization algorithm for training.	categorical	{'adam', 'rmsprop', 'sgd'}
RF	n_estimators	Number of trees in the Random Forest.	int	[50, 200]
	max_depth	Maximum depth of each tree in the Random Forest.	int	[5, 20]

to assess computational efficiency, recognizing that the proposed method achieves superior performance compared to existing models. In Study 2, we focused only on the ROC-AUC metric to evaluate model performance, reaffirming the metric's utility in assessing classification accuracy. These performance metrics collectively provide an objective and thorough evaluation of the models, supporting the effectiveness of the proposed techniques for SDP. Section 2.7 provides a comprehensive account of

ROC-AUC and MCC performance metrics.

7.3.6 Statistical tools

In analysing the effect of hyperparameter tuning (HPT) on software defect prediction (SDP), a statistical test plays a crucial role by providing a rigorous means of assessing the significance of observed differences between models. By applying statistical tests, such as the Friedman test, researchers can determine whether changes in model performance, after applying HPT, are statistically significant rather than due to random variation. This is essential for validating that HPT genuinely improves SDP model outcomes and that observed improvements aren't merely coincidental. In this context, statistical tests allow for an objective comparison across models, classifiers, or techniques, reinforcing the reliability and validity of the findings.

In both Study 1 and Study 2, the Friedman test, a non-parametric statistical tool, is utilized to assess statistically significant differences across multiple groups. In Study 1, the Friedman test is applied to evaluate the performance of various models and classifiers in software defect prediction (SDP), specifically analyzing different models based on resampling and HPT techniques for imbalanced datasets. This allows for a robust comparison of the models' effectiveness in handling such data. In Study 2, the test is used to compare the mean ranks of four distinct techniques following hyperparameter tuning (HPT), enabling an objective assessment of their relative performance. Section 2.7 of Chapter 2 offers a detailed explanation of Friedman test.

7.4 Results Analysis

The result analysis section of this chapter outlines the findings from our empirical investigation into the impact of automated hyperparameter tuning (HPT) using Optuna within the scope of software defect prediction (SDP) models. It provides an in-depth analysis of improvements in predictive performance, the effectiveness of various learning algorithms, and the time-space efficiency achieved through HPT by assessing ROC-AUC and MCC values across multiple datasets and classifiers. The findings from Study 1 offer a comprehensive view of how automated parameter optimization enhances the effectiveness of SMOTE in addressing class imbalance in SDP. Meanwhile, the results for Study 2 showcase the performance of our proposed ensemble-based deep learning (DL) model for SDP.

7.4.1 Results analysis of Study 1

Table 3-7 provides a tabular representation of the impact of HPT on the performance of the SMOTE in the context of SDP models. The tables categorize differences in ROC-AUC and MCC between the default or untuned SMOTE (S), SMOTETUNED (ST), i.e. SMOTE tuned with DE, the proposed approach i.e. optimized OpTunedSMOTE (OS) and the BOTH in which SMOTE, as well as the classifiers, are tuned using OPTUNA, presenting a comprehensive overview of the observed improvements across multiple classifiers and datasets. The analysis is performed assuming SMOTE as the baseline method.

7.4.1.1 Results analysis based on RQ1

RQ1: How does hyperparameter tuning (HPT) of SMOTE impact the predictive performance of software defect prediction (SDP) models across various datasets and classifiers?

Considering the ROC-AUC values presented in Table 7.3 for the KNN classifier, the SMOTETUNED approach shows an average improvement across datasets. The biggest improvements are seen in Debug (18.68%) and JM1 (11.68%). It provides better ROC-AUC scores than SMOTE but doesn't always achieve the highest gains compared to the proposed approach. OpTunedSMOTE shows substantial improvements, particularly in datasets like SWT (9.53%) and Derby-10.2.1.6 (19.43%). It consistently provides higher percentage improvements over SMOTE, indicating it generally enhances model performance more effectively. Improvements shown by BOTH vary and are generally lower compared to OpTunedSMOTE, but they are still positive. The highest gains are observed in SWT (7.35%) and Derby-10.2.1.6 (2.83%). Tuning both the SMOTE as well as the classifier provides competitive results but is not always better than individual optimized methods.

In the case of MCC values, SMOTETUNED shows significant improvements, with the highest being Camel-1.2 (98.9%) and PDE (96.12%). The gains are notable, especially for datasets with initially low MCC scores. OpTunedSMOTE also provides substantial improvements, with the highest in PDE (105.83%) and PC5 (86.58%). It generally shows better performance improvements than SMOTETUNED in MCC scores, indicating it might offer a more effective balance in classifying both classes. BOTH provide varied improvements, with significant gains in SWT (28.37%) and PC5 (32.89%). While effective, the gains are generally lower compared to the best-performing single methods.

Table 7.4, presents the case of the MLP classifier, considering the measure of ROC-AUC, the highest improvement in ROC-AUC is shown in JM1 (35.65%), indicating that SMOTETUNED effectively enhances the model's ability to distinguish between classes. The minimal improvement in Xalan-2.6 (2.48%) suggests that SMOTETUNED has a limited impact on this dataset. OpTunedSMOTE leads to the biggest improvement in PC5 (57.47%) and the least in SWT (7.87%), indicating that this advanced tuning technique significantly boosts the model's performance. BOTH provides the highest notable improvement in ROC-AUC for Prop-3 (15.79%). No improvement is observed in Prop-1 (0.00%), indicating that BOTH does not provide

Table 7.3: Performance metrics of the KNN algorithm across various datasets.

Dataset	**ROC-AUC**				**MCC**			
	S	**ST**	**OS**	**BOTH**	**S**	**ST**	**OS**	**BOTH**
Camel-1.2	0.569	0.578	0.623	0.607	0.091	0.181	0.195	0.093
Debug	0.654	0.776	0.799	0.665	0.200	0.305	0.325	0.318
Derby-10.2.1.6	0.672	0.755	0.802	0.691	0.252	0.357	0.370	0.296
Derby-10.3.1.4	0.670	0.751	0.785	0.750	0.259	0.358	0.378	0.335
JDT	0.815	0.867	0.910	0.822	0.497	0.550	0.563	0.540
JM1	0.592	0.661	0.707	0.697	0.123	0.165	0.179	0.152
ML	0.696	0.775	0.798	0.735	0.251	0.284	0.295	0.280
PC5	0.646	0.741	0.768	0.676	0.149	0.259	0.278	0.198
PDE	0.682	0.739	0.775	0.724	0.103	0.202	0.212	0.154
Prop-1	0.743	0.757	0.790	0.752	0.291	0.321	0.332	0.316
Prop-2	0.729	0.737	0.786	0.777	0.192	0.262	0.276	0.268
Prop-3	0.589	0.683	0.731	0.627	0.113	0.129	0.148	0.140
Prop-4	0.632	0.707	0.755	0.685	0.118	0.175	0.185	0.153
Prop-5	0.659	0.700	0.741	0.720	0.074	0.098	0.114	0.077
SWT	0.872	0.909	0.955	0.936	0.601	0.782	0.798	0.771
Xalan-2.5	0.719	0.765	0.805	0.758	0.329	0.378	0.391	0.355
Xalan-2.6	0.782	0.831	0.854	0.817	0.401	0.492	0.507	0.472

any additional benefit over the baseline SMOTE. Considering the MCC values, the highest improvement in JM1 (702.74%) is shown by SMOTETUNED, suggesting that JM1 has a complex imbalance that is well addressed by tuning SMOTE parameters, while the least improvement is achieved in Xalan-2.6 (5.77%). The highest improvement in JM1 (720.55%) and lowest improvement in Prop-3 (18.67%) with OpTunedSMOTE further underscores the dataset's complexity and the effectiveness of optimal parameter tuning. For BOTH, there is a normal improvement range varying from Xalan-2.6 (5.13%) to Prop-2 (6.43%).

In Table 7.5, for the RF classifier, considering ROC-AUC, SMOTETUNED shows improvements in ROC-AUC ranging from slight decreases to moderate increases, with most datasets showing positive improvements such as in Derby-10.3.1.4 (10.04%) and Derby-10.2.1.6 (6.78%). OpTunedSMOTE consistently shows a higher percentage improvement in ROC-AUC compared to default SMOTE, with some datasets showing

Table 7.4: Performance metrics of MLP algorithm across various datasets

Dataset	**ROC-AUC**				**MCC**			
	S	**ST**	**OS**	**BOTH**	**S**	**ST**	**OS**	**BOTH**
Camel-1.2	0.599	0.628	0.660	0.638	0.153	0.251	0.264	0.212
Debug	0.705	0.767	0.806	0.705	0.237	0.358	0.372	0.290
Derby-10.2.1.6	0.651	0.732	0.754	0.663	0.252	0.421	0.432	0.405
Derby-10.3.1.4	0.619	0.749	0.783	0.662	0.250	0.401	0.420	0.266
JDT	0.770	0.860	0.908	0.822	0.474	0.548	0.568	0.540
JM1	0.460	0.624	0.653	0.557	0.073	0.586	0.599	0.084
ML	0.762	0.818	0.851	0.789	0.299	0.349	0.365	0.331
PC5	0.482	0.719	0.759	0.534	0.106	0.251	0.267	0.108
PDE	0.713	0.747	0.767	0.761	0.225	0.292	0.307	0.243
Prop-1	0.705	0.762	0.807	0.705	0.160	0.282	0.300	0.278
Prop-2	0.703	0.758	0.806	0.773	0.171	0.186	0.203	0.182
Prop-3	0.633	0.719	0.758	0.733	0.075	0.136	0.153	0.089
Prop-4	0.676	0.746	0.780	0.690	0.172	0.214	0.230	0.207
Prop-5	0.673	0.721	0.752	0.748	0.109	0.157	0.174	0.166
SWT	0.865	0.885	0.933	0.878	0.545	0.792	0.803	0.548
Xalan-2.5	0.683	0.744	0.775	0.733	0.225	0.346	0.363	0.294
Xalan-2.6	0.808	0.828	0.867	0.810	0.468	0.495	0.511	0.492

significant improvements. The highest improvements are shown in Derby-10.3.1.4 (15.19%) and Debug (10.00%). When both SMOTE and classifiers are tuned, improvements in ROC-AUC are generally positive and sometimes slightly lower than OpTunedSMOTE alone. For example, Derby-10.3.1.4 shows a 6.83% improvement, and Prop-1 shows a 5.46% improvement. In the case of MCC values, SMOTE-TUNED typically shows a positive percentage improvement in MCC, with substantial gains in some datasets. Notable improvements include Derby-10.3.1.4 (60.27%) and Camel-1.2 (71.29%). OpTunedSMOTE consistently provides higher percentage improvements in MCC compared to default SMOTE. The highest improvements are seen in datasets like Derby-10.3.1.4 (66.78%) and Prop-2 (45.91%). BOTH generally results in positive percentage improvements in MCC, often comparable to OS alone. Significant improvements are observed in datasets like Derby-10.3.1.4 (19.86%) and Prop-2 (43.59%).

Table 7.5: Performance metrics of RF algorithm across various datasets

Dataset	ROC-AUC				MCC			
	S	ST	O	BOTH	S	ST	O	BOTH
Camel-1.2	0.653	0.652	0.695	0.671	0.202	0.346	0.366	0.244
Debug	0.760	0.789	0.836	0.789	0.325	0.292	0.331	0.308
Derby-10.2.1.6	0.723	0.772	0.794	0.735	0.356	0.457	0.470	0.460
Derby-10.3.1.4	0.717	0.789	0.826	0.766	0.292	0.468	0.487	0.350
JDT	0.867	0.884	0.908	0.902	0.550	0.574	0.586	0.563
JM1	0.674	0.700	0.732	0.729	0.218	0.247	0.266	0.235
ML	0.820	0.835	0.878	0.840	0.344	0.386	0.400	0.392
PC5	0.778	0.835	0.870	0.833	0.378	0.386	0.399	0.385
PDE	0.762	0.774	0.804	0.792	0.293	0.298	0.315	0.308
Prop-1	0.787	0.784	0.832	0.830	0.237	0.332	0.346	0.263
Prop-2	0.738	0.787	0.837	0.768	0.257	0.356	0.375	0.369
Prop-3	0.709	0.723	0.749	0.712	0.110	0.152	0.168	0.114
Prop-4	0.739	0.763	0.806	0.799	0.168	0.238	0.249	0.203
Prop-5	0.724	0.732	0.777	0.749	0.107	0.131	0.150	0.120
SWT	0.920	0.935	0.961	0.933	0.692	0.838	0.855	0.732
Xalan-2.5	0.775	0.784	0.814	0.808	0.419	0.454	0.467	0.429
Xalan-2.6	0.848	0.859	0.909	0.876	0.540	0.549	0.564	0.546

Table 7.6: Performance metrics of SVM algorithm across various datasets.

Dataset	ROC-AUC				MCC			
	S	ST	OS	BOTH	S	ST	OS	BOTH
Camel-1.2	0.543	0.651	0.681	0.638	0.064	0.227	0.244	0.210
Debug	0.615	0.724	0.771	0.770	0.188	0.246	0.260	0.194
Derby-10.2.1.6	0.497	0.711	0.755	0.709	0.083	0.100	0.112	0.087
Derby-10.3.1.4	0.505	0.595	0.624	0.538	0.018	0.045	0.064	0.021
JDT	0.783	0.810	0.843	0.823	0.332	0.335	0.350	0.341
JM1	0.542	0.668	0.717	0.559	0.519	0.523	0.541	0.536
ML	0.612	0.691	0.733	0.706	0.082	0.085	0.099	0.096
PC5	0.615	0.691	0.713	0.711	0.188	0.220	0.236	0.194
PDE	0.651	0.700	0.721	0.679	0.016	0.141	0.160	0.138
Prop-1	0.564	0.616	0.645	0.634	0.154	0.294	0.311	0.168
Prop-2	0.544	0.562	0.588	0.558	0.069	0.056	0.072	0.067
Prop-3	0.523	0.558	0.587	0.539	0.096	0.110	0.122	0.117
Prop-4	0.525	0.631	0.656	0.611	0.071	0.162	0.176	0.076
Prop-5	0.559	0.626	0.672	0.646	0.123	0.143	0.156	0.128
SWT	0.844	0.856	0.877	0.846	0.554	0.689	0.699	0.689
Xalan-2.5	0.659	0.670	0.692	0.685	0.219	0.248	0.267	0.244
Xalan-2.6	0.785	0.792	0.835	0.828	0.387	0.394	0.411	0.408

In Table 7.6, for the SVM classifier, SMOTETUNED shows improvements in ROC-AUC, ranging from slight increases to significant increases, with most datasets showing positive improvements. Notable improvements include Derby-10.2.1.6 (43.06%) and Camel-1.2 (19.89%). Consistently, there is a higher percentage of improvements in ROC-AUC than in default SMOTE. Significant improvements are observed in datasets such as Derby-10.2.1.6 (51.90%) and Camel-1.2 (25.41%). BOTH generally leads to positive improvements in ROC-AUC, sometimes slightly lower than OpTunedSMOTE alone. For example, Debug shows a 25.20% improvement, and Camel-1.2 shows a 17.51% improvement. In the case of MCC, SMOTETUNED typically shows positive percentage improvements in MCC, with substantial gains in some datasets. Notable improvements include Camel-1.2 (254.69%) and PDE (781.25%). OpTunedSMOTE consistently provides higher percentage improvements in MCC compared to default SMOTE. Highest improvements are observed in datasets like PDE (900.00%) and Camel-1.2 (281.25%). BOTH results in positive percentage improvements in MCC, often comparable to OpTunedSMOTE alone. Significant improvements are observed in datasets like PDE (762.50%) and Camel-1.2 (228.13%).

In Table 7.7, for the XGB classifier, SMOTETUNED shows improvements in ROC-AUC ranging from slight increases to significant increases, with most datasets showing positive improvements. Notable improvements include Camel-1.2 (29.38%) and PDE (26.53%). OpTunedSMOTE consistently shows higher percentage improvements in ROC-AUC compared to default SMOTE. Significant improvements are observed in datasets such as Camel-1.2 (38.26%) and PDE (32.20%). BOTH show positive improvements in ROC-AUC, sometimes slightly lower than OpTunedSMOTE alone. For example, Debug shows a 5.98% improvement, and Camel-1.2 shows a 4.74% improvement. SMOTETUNED typically shows positive percentage improvements in MCC, with substantial gains in some datasets. Notable improvements include SWT (22.04%) and Derby-10.3.1.4 (48.75%). OpTunedSMOTE consistently pro-

Table 7.7: Performance metrics of XGB algorithm across various datasets.

Dataset	**ROC-AUC**				**MCC**			
	S	**ST**	**OS**	**BOTH**	**S**	**ST**	**OS**	**BOTH**
Camel-1.2	0.549	0.710	0.759	0.575	0.306	0.343	0.356	0.335
Debug	0.786	0.800	0.837	0.833	0.304	0.426	0.443	0.311
Derby-10.2.1.6	0.762	0.779	0.812	0.801	0.364	0.470	0.488	0.444
Derby-10.3.1.4	0.756	0.783	0.814	0.764	0.320	0.476	0.495	0.353
JDT	0.881	0.889	0.911	0.900	0.527	0.541	0.556	0.556
JM1	0.699	0.709	0.729	0.714	0.250	0.250	0.269	0.264
ML	0.828	0.843	0.867	0.833	0.317	0.356	0.376	0.333
PC5	0.796	0.802	0.843	0.835	0.373	0.387	0.403	0.396
PDE	0.634	0.802	0.838	0.707	0.291	0.299	0.318	0.295
Prop-1	0.786	0.804	0.843	0.820	0.232	0.317	0.333	0.271
Prop-2	0.838	0.853	0.896	0.843	0.313	0.350	0.363	0.326
Prop-3	0.681	0.742	0.765	0.690	0.130	0.136	0.152	0.151
Prop-4	0.681	0.766	0.807	0.748	0.138	0.155	0.168	0.149
Prop-5	0.728	0.760	0.791	0.749	0.093	0.100	0.111	0.100
SWT	0.919	0.925	0.971	0.920	0.685	0.836	0.853	0.790
Xalan-2.5	0.784	0.791	0.812	0.789	0.406	0.448	0.467	0.455
Xalan-2.6	0.858	0.867	0.893	0.858	0.568	0.586	0.601	0.585

vides higher percentage improvements in MCC compared to default SMOTE. The highest improvements are observed in datasets like Derby-10.3.1.4 (54.69%) and Debug (45.72%). BOTH results in positive percentage improvements in MCC, often comparable to OpTunedSMOTE alone. Significant improvements are observed in datasets like Derby-10.3.1.4 (10.31%) and SWT (15.33%).

7.4.1.2 Results analysis based on RQ2

RQ2: Which classifiers benefit the most from optimized SMOTE techniques (SMOTE-TUNED, OpTunedSMOTE, and BOTH) in terms of predictive accuracy and robustness?

[h]

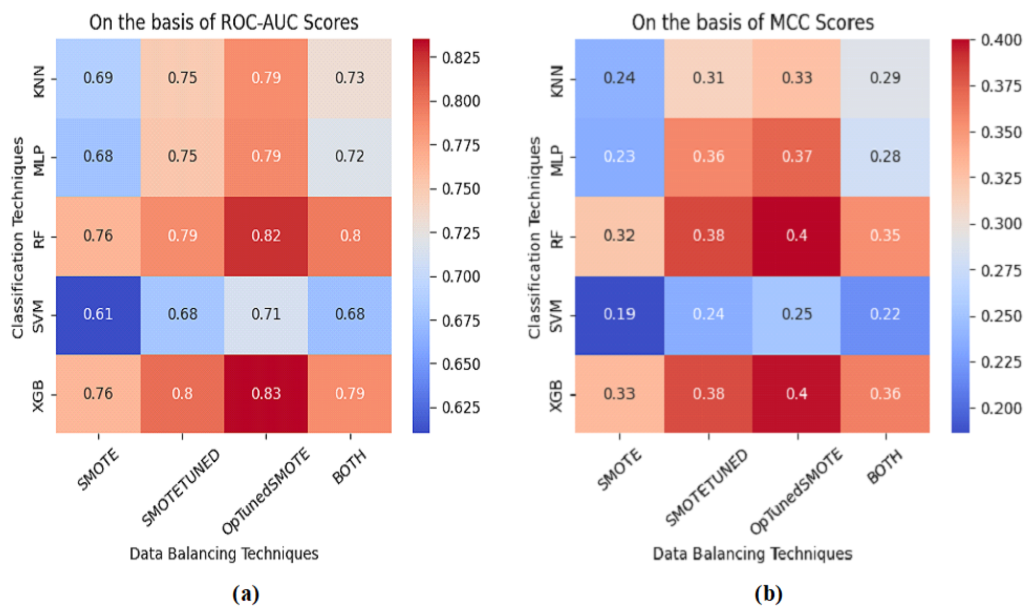


Figure 7.4: Average performance of different classification techniques with various data balancing techniques (a) on the basis of ROC-AUC values and (b) on the basis of MCC values.

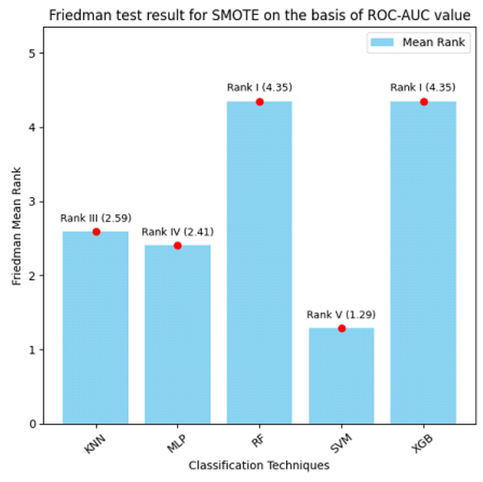
Considering the average performances of the classification techniques for 17 datasets in Figure 7.4(a), KNN benefits substantially from hyperparameter tuning, with OpTunedSMOTE leading to the highest ROC-AUC scores of 0.79 compared to other data balancing techniques. MLP shows a significant gain in ROC-AUC (0.79), indicating it performs well with enhanced data balancing techniques like OpTunedSMOTE. RF consistently achieves high ROC-AUC scores (0.82), particularly with OpTunedSMOTE, highlighting its robustness and effectiveness. SVM shows moderate improvements, with the highest gains seen with OpTunedSMOTE in terms of ROC-AUC score (0.71). XGB stands out with the highest ROC-AUC scores (0.83), particularly with OpTunedSMOTE, indicating it excels with hyperparameter-tuned SMOTE.

From Figure 7.4(b), it can be concluded that KNN shows moderate improvements in the average value of MCC (0.33) with hyperparameter tuning, particularly with OpTunedSMOTE and SMOTETUNED. MLP shows a notable increase in MCC (0.37) with OpTunedSMOTE, indicating it responds well to these enhancements. RF demonstrates a robust performance boost from hyperparameter tuning, achieving the highest MCC (0.40) among all classifiers. While SVM benefits from OpTunedSMOTE, its highest MCC values (0.25) are generally lower compared to other classifiers, indicating it may not be the best choice for these datasets. XGB shows strong performance gains with hyperparameter tuning, making it one of the top-performing classifiers in terms of MCC (0.40).

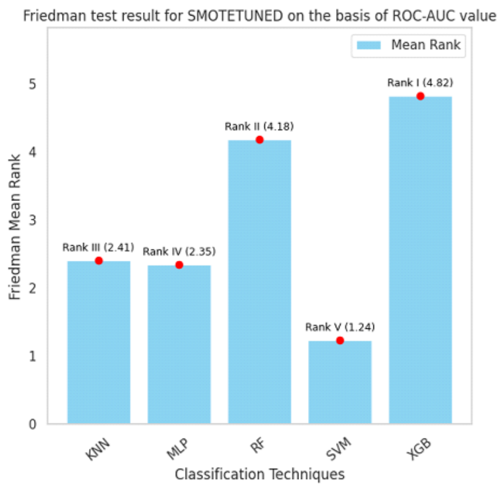
7.4.1.3 Results analysis based on RQ3

RQ3: What are the statistically significant differences among classification methods following hyperparameter-tuned SMOTE in SDP models?

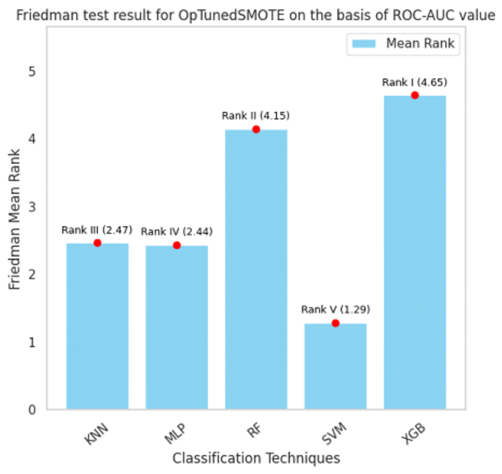
[h]



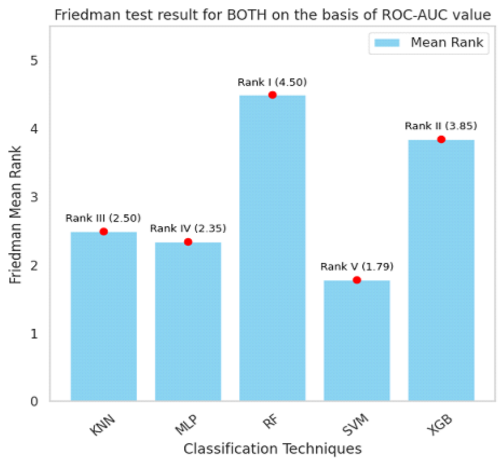
(a)



(b)



(c)



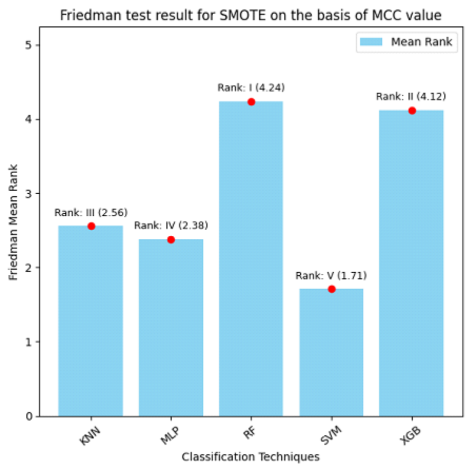
(d)

Figure 7.5: Friedman test results for classification techniques based on ROC-AUC values (a) for SMOTE techniques (b) for SMOTETUNED technique (c) for OpTunedSMOTE technique (d) for BOTH techniques

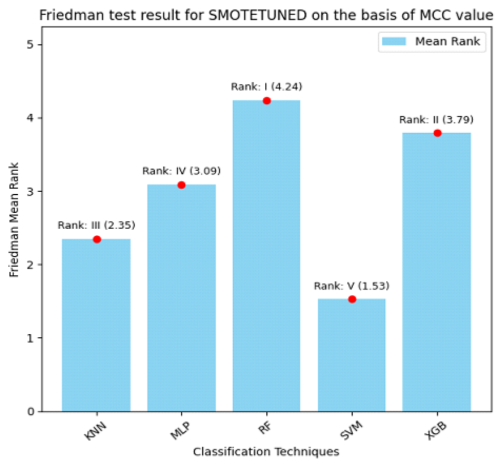
The Friedman test results provide a comparative ranking of various classification techniques (KNN, MLP, RF, SVM, and XGB) based on their ROC-AUC values under different data balancing techniques (SMOTE, SMOTETUNED, OpTunedSMOTE, and BOTH).

From Figure 7.5(a), XGB and RF are the top-performing classifiers with the highest mean ranks (4.35), indicating they achieve the highest ROC-AUC values using default SMOTE. SVM ranks the lowest (1.29), showing it performs the worst in terms of ROC-AUC with default SMOTE. KNN and MLP have moderate ranks, suggesting average performance with default SMOTE. Figure 7.5(b) shows the case for SMOTETUNED and reveals that XGB continues to be the best performer with the highest mean rank (4.82) when SMOTE is tuned with DE, followed by RF (4.18). SVM remains the lowest (1.24), indicating it still performs poorly even with SMOTETUNED. KNN and MLP show slight improvements but remain in moderate ranks. Depicting the case of OpTunedSMOTE, Figure 7.5(c), XGB and RF continue to dominate with the highest ranks (4.65 and 4.15, respectively) under OpTunedSMOTE, indicating their strong performance. SVM again ranks the lowest (1.29), showing little improvement. KNN and MLP remain in the middle, with no significant changes in rank. Figure 7.5(d) depicts the case of BOTH, in which RF takes the top spot (4.50) when both SMOTE and classifiers are tuned, followed by XGB (3.85). SVM, although showing some improvement, still ranks the lowest (1.79). KNN and MLP have consistent moderate performance with slight variations. XGB and RF consistently rank the highest across all tuning methods. The rankings of XGB and RF remain at the top before and after tuning, indicating their robustness and substantial improvement with SMOTE tuning. Both classifiers show the most significant performance gains with OpTunedSMOTE, reinforcing the effectiveness of combined tuning. The rankings of KNN and MLP remain relatively stable, showing moderate improvements with each tuning method. Their ranks do not change drastically, suggesting they benefit from tuning but are not as sensitive to it as XGB and RF. SVM consistently ranks the lowest across all methods. While there are slight improvements with tuning, its rank remains the lowest, indicating limited benefit from SMOTE tuning and the potential need for further optimization or alternative techniques.

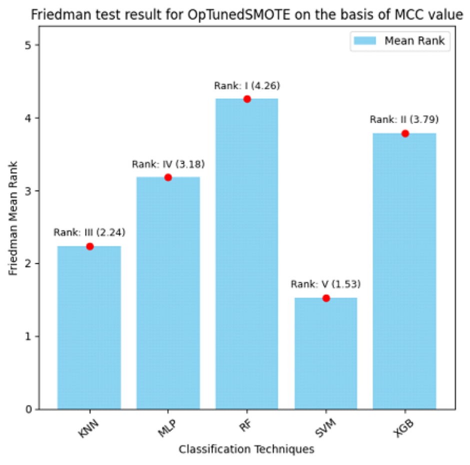
[h]



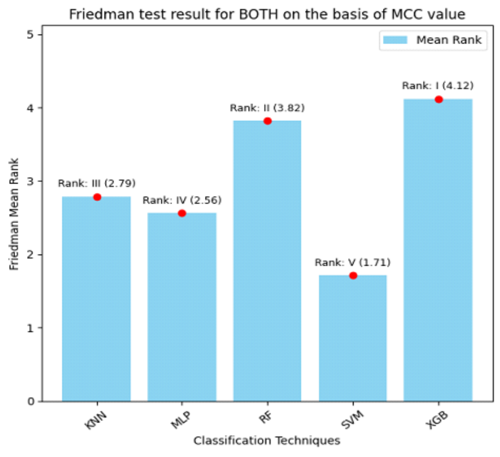
(a)



(b)



(c)



(d)

Figure 7.6: Friedman test results for classification techniques based on MCC values (a) for SMOTE techniques (b) for SMOTETUNED technique (c) for OpTunedSMOTE technique (d) for BOTH technique

From Figure 7.6(a), it can be analyzed that in the case of default SMOTE, RF is the top performer with the highest mean rank (4.24), followed closely by XGB (4.12). SVM ranks the lowest (1.71), showing it performs the worst in terms of MCC with default SMOTE. KNN and MLP have moderate ranks, indicating average performance with default SMOTE. Figure 7.6(b) reveals that RF maintains its top position (4.24) with SMOTETUNED, followed by XGB (3.79). SVM remains the lowest (1.53), indicating it still performs poorly even with SMOTETUNED. KNN shows slight improvement, while MLP's rank decreases slightly. In Figure 7.6(c), RF continues to dominate (4.26) under OpTunedSMOTE, with XGB also performing well (3.79). SVM remains the lowest (1.53), showing little improvement. KNN shows the best performance improvement, moving to Rank II. In the case of BOTH from Figure 7.6(d), RF takes the top position (3.82) when both SMOTE and classifiers are tuned, with XGB closely following (4.12). SVM, although showing some improvement, still ranks the lowest (1.71). KNN and MLP maintain moderate performance with slight variations.

7.4.1.4 Results analysis based on RQ4

RQ4: How does hyperparameter tuning of SMOTE affect the computational resources required, such as memory usage and execution time, for different classifiers?

[h]

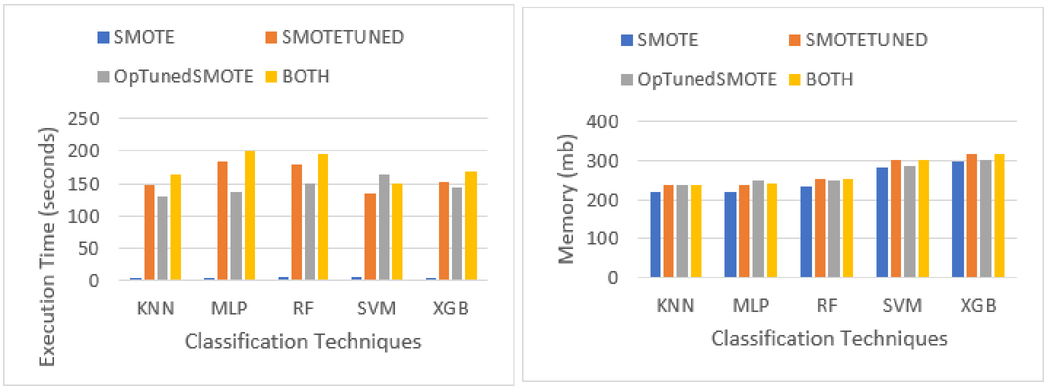


Figure 7.7: Comparison of execution time and memory usage for different classification techniques with various SMOTE variations.

Based on Figure 7.7, KNN shows a slight increase in memory usage with SMOTETUNED (237.10 MB) and OpTunedSMOTE (239.11 MB), with the highest usage observed in BOTH (239.38 MB). MLP's memory usage increases moderately with tuning, with BOTH (247.54 MB) showing the highest usage. RF shows a significant increase in memory usage with tuning methods, with a slight decrease in BOTH (247.42 MB) compared to SMOTETUNED (252.43 MB) and OpTunedSMOTE (252.01 MB). SVM shows significant increases with tuning, except for BOTH (256.13 MB), which slightly reduces memory usage. XGB's memory usage increases with SMOTETUNED (315.87 MB) and OpTunedSMOTE (314.69 MB) but significantly decreases with BOTH (250.83 MB), indicating a trade-off optimization. Memory usage increases for all classifiers when moving from SMOTE to SMOTETUNED, OpTunedSMOTE, and BOTH. BOTH generally shows the highest memory usage, indicating the most computationally intensive method.

In the case of execution time, KNN's execution time increases significantly with tuning methods, particularly with BOTH (162.88 sec). MLP shows the highest increase in execution time with SMOTETUNED (184.46 sec) and BOTH (199.87 sec), with OpTunedSMOTE being slightly lower (135.83 sec). RF's execution time increases significantly with all tuning methods, particularly with BOTH (194.3 sec). SVM's execution time increases significantly with tuning, with OpTunedSMOTE (163.48 sec) being the most time-intensive. XGB shows significant increases in execution time with all tuning methods, with BOTH (168.20 sec) having the highest execution time. Execution time increases significantly for all classifiers with SMOTETUNED, OpTunedSMOTE, and BOTH compared to SMOTE. BOTH consistently results in the highest execution time, indicating a significant trade-off for improved performance.

Our findings highlight the importance of parameter settings in defect prediction models and advocate for the use of automated hyperparameter optimization techniques to fully utilize ML classifiers in software development processes. The findings show that HPT of SMOTE improves the predictive performance of software development process models. The use of Optuna for optimization resulted in significant improvements in the models. The study also explores the comparative effectiveness of different learning algorithms after HPT. The time and space efficiency of the HPT process is demonstrated, with Optuna outperforming other techniques. The study suggests that off-the-shelf configurations for data balancing techniques may not be advisable; HPT is required for optimal performance.

7.4.2 Results analysis of Study 2

This section presents the outcomes of our study on SDP using DL techniques and proposed model based on stacked ensemble model.

7.4.2.1 Results analysis based on RQ1

RQ1 How does HPT affect the performance of different DL techniques in terms of AUC improvement?

The study evaluated the impact of hyperparameter tuning (HPT) on the performance of various deep learning (DL) techniques, specifically focusing on the percentage improvement in AUC. The results, summarized in Figure 7.8 and detailed in Table 7.8, provide a comprehensive comparison of the performance gains across different models.

Figure 7.8 visually represents the average percentage improvement in AUC for techniques before HPT and after HPT, clearly illustrating that CNN benefited the most from HPT with an impressive average improvement of 20.87%. This substantial gain indicates that CNNs are highly sensitive to hyperparameter adjustments, leading to significant performance enhancements when appropriately tuned. LSTM also exhibited considerable gains, with an average AUC improvement of 19.75%, as shown in Table 1. This close performance to CNN suggests that LSTM methods also gain significantly from hyperparameter tuning, though slightly less than CNN.

GRU demonstrated a moderate improvement of 15.05% in AUC, as presented in both Figure 7.8 and Table 7.8. While GRUs benefit from hyperparameter tuning, the extent of improvement is less pronounced compared to CNN and LSTM. This moderate gain suggests that GRUs are somewhat less sensitive to hyperparameter changes. Lastly, EL methods showed the smallest improvement at 6.74%, indicating a relative robustness to hyperparameter adjustments. As presented in Figure 2, this minimal improvement suggests that EL models may perform adequately even with default hyperparameters, implying a lower sensitivity to tuning.

Table 7.8: ROC-AUC Scores for CNN, LSTM, GRU, and ENSEMBLE Models With and Without HPT

Datasets	**CNN**		**LSTM**		**GRU**		**ENSEMBLE**	
	W/O HPT	**With HPT**	**W/O HPT**	**With HPT**	**W/O HPT**	**With HPT**	**W/O HPT**	**With HPT**
Ant-1.7	0.779	0.839	0.682	0.824	0.758	0.825	0.835	0.880
Camel-1.6	0.688	0.717	0.642	0.682	0.580	0.661	0.679	0.737
Ivy-2.0	0.620	0.871	0.714	0.819	0.639	0.787	0.845	0.893
Jedit-4.3	0.644	0.914	0.751	0.830	0.632	0.753	0.901	0.953
Log4j-1.2	0.687	0.848	0.685	0.786	0.608	0.743	0.850	0.932
Xerces-1.4	0.859	0.923	0.838	0.873	0.877	0.903	0.903	0.952

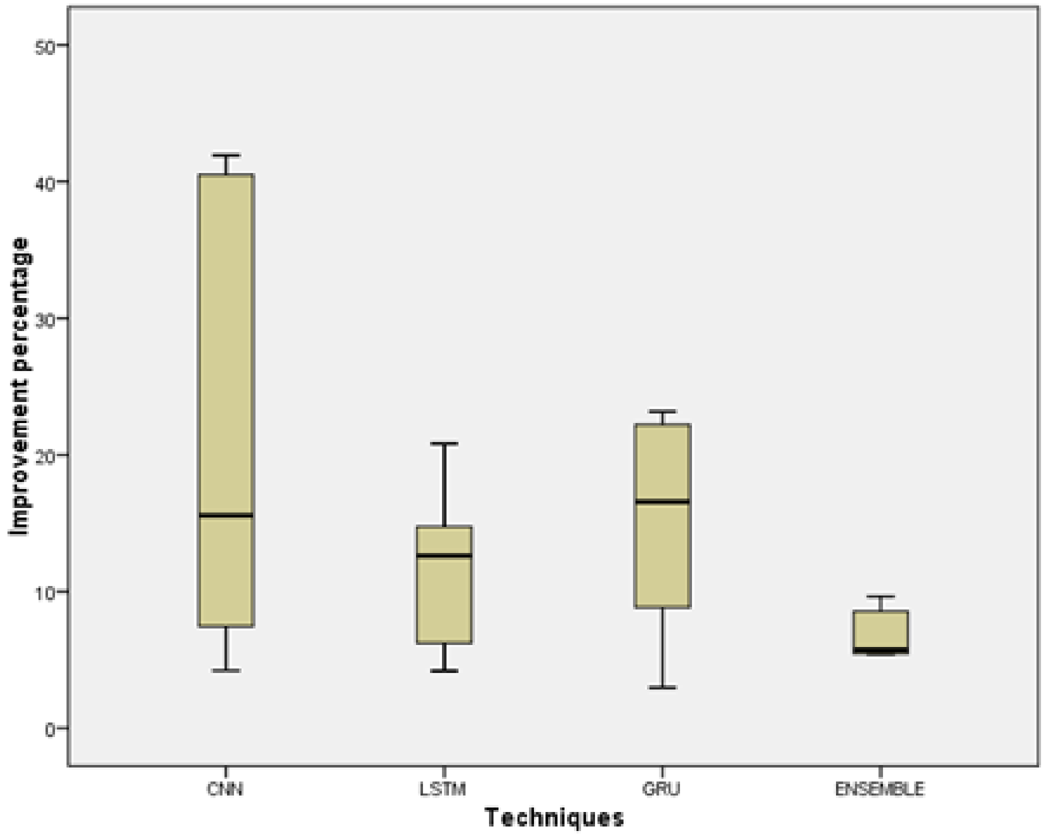


Figure 7.8: Percentage improvement in ROC-AUC for each technique.

These findings highlight the varying degrees to which different models rely on HPT for optimal performance, with CNNs and LSTMs showing the greatest sensitivity and potential for enhancement through fine-tuning. The visual and tabular data collectively emphasize the importance of model-specific tuning strategies to achieve optimal performance in DL applications.

7.4.2.2 Results analysis based on RQ2

RQ2 Which DL method demonstrates the highest performance in predicting software defects in this study?

This study employed Friedman's test to evaluate the mean rank of four different techniques after HPT. The mean ranks obtained are depicted in Table 7.9. The test yielded a p-value of 0.001 at a 5% level of significance. The p-value indicates a statistically significant difference in the performance of these techniques.

Table 7.9: Mean rank of various techniques with HPT

Techniques	**Mean Rank (p = 0.001)**
CNN	2.00 (II)
LSTM	3.33 (III)
GRU	3.67 (IV)
ENSEMBLE	1.00 (I)

Among these, the Ensemble Learning (EL) technique demonstrated the most promising results, achieving the first rank (I). This consistent superiority of the EL technique across various datasets suggests that it outperforms the other evaluated techniques. The enhanced performance of the EL technique can be attributed to its ability to amalgamate the strengths of multiple techniques, thereby enhancing predictive accuracy and robustness. Consequently, within the scope of this analysis, the EL technique emerges as the most effective technique.

In comparison, the other techniques also exhibited noteworthy performances. With a mean rank of 2.00, the CNN technique performed relatively well, indicating a solid predictive capability after hyperparameter tuning. The LSTM technique, with a mean rank of 3.33, acquired the III rank and showed moderate performance. Although it did not outperform the EL or CNN techniques, LSTM's ability to capture long-term dependencies makes it a viable option for time-series data and sequential predictions.

The GRU technique had the lowest performance among the techniques with a mean rank of 3.67.

Overall, while Ensemble Learning stands out as the most promising technique, CNNs and LSTM techniques also show substantial potential in specific applications. GRUs, though ranked lowest, still provide a valuable option depending on the context of the problem. The statistical significance of the Friedman's test reinforces the reliability of these findings.

7.5 Discussion

This chapter highlights the significant impact of automated hyperparameter tuning (HPT) using Optuna on enhancing Software Defect Prediction (SDP) models across Machine Learning (ML) and Deep Learning (DL) frameworks. Both studies demonstrate that HPT markedly improves predictive performance, with tailored parameter adjustments in SMOTE and DL models enhancing key performance metrics like ROC-AUC and MCC across diverse datasets.

In Study 1, the OpTunedSMOTE approach consistently demonstrated superior performance over default SMOTE and SMOTETUNED across all evaluated classifiers. By tuning SMOTE's parameters to adapt to the characteristics of each dataset, OpTunedSMOTE achieved the highest improvements in ROC-AUC and MCC scores, indicating more effective handling of class imbalance and better overall predictive accuracy. Random Forest (RF) and Extreme Gradient Boosting (XGB) emerged as the top-performing classifiers, showing the most significant gains from OpTunedSMOTE. These classifiers, known for their robustness and adaptability, benefited greatly from HPT, suggesting that automated parameter optimization can maximize their performance in defect prediction tasks. Although OpTunedSMOTE yielded notable improvements in performance, it came with increased computational costs, especially in terms of memory usage and execution time. This trade-off implies that while OpTunedSMOTE is highly effective, it may not be ideal for resource-constrained environments. For such cases, SMOTETUNED offers a balanced option, providing moderate improvements with lower computational demands. The comparative analysis in Study 1 employed the Friedman test to statistically validate the effectiveness of different tuning approaches. This analysis confirmed the superiority of OpTunedSMOTE over other methods, with the highest rankings across various classifiers.

Study 2 showed that Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) models are particularly sensitive to HPT, resulting in AUC improvements of 20.87% and 19.75%, respectively. These results indicate that CNNs and LSTMs benefit significantly from tuning, making HPT a crucial step for applications involving spatial or sequential data in SDP tasks. While Gated Recurrent Units (GRUs) showed moderate AUC improvements of 15.05%, they appeared less sensitive to hyperparameter changes compared to CNNs and LSTMs. This finding suggests that GRUs could be more suitable for applications where computational efficiency is prioritized over maximum performance, offering a good balance between accuracy and resource consumption. The study also found that Ensemble Learning (EL) models showed the least sensitivity to HPT, with an AUC improvement of only 6.74%. This robustness to hyperparameter adjustments suggests that EL models can achieve satisfactory performance with default settings, making them suitable for resource-limited environments.

The findings from both studies underscore the importance of selecting the appropriate model and tuning approach based on the specific requirements of the SDP task. OpTunedSMOTE and HPT-enhanced CNNs and LSTMs are recommended for tasks requiring high accuracy and capable of handling computational costs. Conversely, GRUs and Ensemble Learning models may be preferred in scenarios where simplicity, robustness, or resource efficiency is essential.

Practitioners are encouraged to prioritize HPT in the model development process to maximize predictive accuracy. For complex and data-intensive tasks, especially in SDP, dedicating resources to explore a range of hyperparameter configurations can yield substantial performance benefits. While OpTunedSMOTE and DL models like CNNs and LSTMs offer maximum performance improvements, practitioners should consider the increased computational requirements. In resource-limited settings, models like GRUs and EL with moderate or default tuning may be preferable for their balance of efficiency and effectiveness.

Chapter 8

Conclusion

8.1 Summary of the Research Work

Software Defect Prediction (SDP) is a critical aspect of the software development lifecycle, enabling early identification of potential issues within code. As software systems grow in complexity, the likelihood of defects increases, necessitating proactive strategies to maintain quality and reliability. The primary aim of the work conducted in this thesis is to develop models that improve the prediction of defect-prone areas, facilitating efficient allocation of limited testing and maintenance resources. Accurate defect prediction allows organizations to prevent costly post-release failures and maintain high customer satisfaction. This thesis proposes and evaluates several techniques to optimize SDP models, ranging from traditional machine learning approaches to advanced metaheuristic optimization and deep learning methods. Given the promising performance of these methods in enhancing predictive accuracy, the work also explores ensemble techniques and automated hyperparameter tuning to further refine results. Through structured empirical experiments, the research presented in this thesis demonstrates the effectiveness of these approaches, providing valuable insights for researchers and practitioners aiming to improve software quality assurance.

To evaluate and assess the current state of literature in the domain of software evolution, we performed a systematic literature review to examine and evaluate existing feature reduction techniques for Software Quality Predictive Modeling (SQPM). Covering research from 2000 to 2019, this study analyzed 22 primary studies and identified commonly used feature selection methods, metrics, datasets, and machine learning techniques in SDP. Techniques were grouped into three main categories: filter, wrapper, and embedded. Frequently used methods included Correlation-based Feature Selection, OneR, and Information Gain, while less common but valuable approaches included Principal Component Analysis (PCA) and Gain Ratio. Additionally, the PROMISE dataset was used in 45% of the studies, while Random Forest, Logistic Regression, and NaÃ¯ve Bayes were the preferred machine learning classifiers in 45% of the studies. ROC-AUC was used as the primary evaluation metric in 63% of the studies, followed by accuracy at 27% and F-measure at 23%. This literature review laid the groundwork by identifying gaps and establishing a baseline for further empirical investigations into feature selection techniques in SDP.

In software defect prediction (SDP), handling high-dimensional data is a critical challenge that can significantly impact the performance of predictive models. Feature extraction techniques are essential for reducing the complexity of datasets while preserving the most relevant information. By providing an in-depth comparison of various feature extraction techniques, we performed a study that significantly contributes to SDP by providing a comparison of feature extraction techniques. In this conducted a detailed empirical analysis of four feature extraction techniques in combination with a Support Vector Machine (SVM) classifier. The study utilized the PROMISE dataset, employing evaluation metrics such as accuracy and ROC-AUC to assess the effectiveness of each feature extraction technique. The results demonstrated that autoencoders achieved the highest ROC-AUC scores, with an improvement of 15%

over other methods, while Linear Discriminant Analysis (LDA) also showed favorable performance with an average improvement of 10% across various datasets. Statistical validation through the non-parametric Friedman test confirmed the reliability of these findings, highlighting autoencoders as a promising method for defect prediction tasks. This study illustrated the importance of feature extraction in improving model accuracy and underscored the potential of deep learning techniques in enhancing SDP models.

In real-world software development, projects often suffer from limited historical data, making it challenging to build effective defect prediction models. Cross-project defect prediction (CPDP) techniques are vital as they allow the use of data from other projects to predict defects in a target project. It is essential to address the need to improve CPDP models, particularly through the integration of feature selection techniques, which can enhance the transferability ty and accuracy of predictions across different projects. For this we performed a study which addresses the challenge of building effective software defect prediction models for projects with limited past data by exploring CPDP techniques. This study compared five Feature Subset Selection (FSS) techniques and five Feature Ranking (FR) methods across multiple datasets using five machine learning classifiers. The results indicated that FSS techniques consistently outperformed FR techniques and non-feature selection methods, with a 12% average improvement in ROC-AUC scores. AdaBoost emerged as the most effective classifier with a 15% improvement in predictive accuracy when paired with FSS-selected features. Additionally, certain FSS methods, such as Best First and Greedy Stepwise, demonstrated consistent feature selection across datasets. Statistical testing confirmed that these FSS techniques could improve CPDP in various scenarios, particularly for datasets with limited historical data, revealing a valuable approach for enhancing defect prediction accuracy by selectively choosing features.

From past studies we can analyse that evolutionary feature selection techniques play a crucial role in software defect prediction by effectively handling high-dimensional data and identifying the most relevant features for prediction accuracy. These techniques, inspired by natural selection processes, iteratively improve feature subsets, eliminating redundant and irrelevant attributes that could hinder model performance. Hence, we introduced a novel two-phase Grey Wolf Optimizer (2M-GWO) combined with a Random Forest classifier to improve feature selection for SDP. Using 27 datasets from diverse sources, the study demonstrated that the 2M-GWO-RF model achieved the highest average ROC-AUC rank with a mean score of 2.19 across datasets, showcasing an improvement of 18% over other feature selection approaches. The two-phase 2M-GWO approach effectively reduced feature dimensionality, with an average reduction of 25% in feature set size, while maintaining high prediction accuracy. Statistical validation with the Friedman test (p = 0.034) confirmed the significance of these results, positioning the 2M-GWO-RF model as a robust feature selection approach for defect prediction. This study underscored the potential of meta-heuristic optimization techniques in refining feature selection to enhance the predictive power of SDP models.

Hyperparameter optimization plays a pivotal role in enhancing the performance and reliability of software defect prediction models, particularly when dealing with challenges like imbalanced data and hyperparameter tuning. While tuning hyperparameters of classifiers improves the model's specific learning patterns, tuned preprocessing optimizes the data foundation, enhancing the classifier's ability to generalize accurately. Together, they create a robust, reliable model, with preprocessing setting the stage for more effective classifier performance. We carried out experiments investigating the effects of hyperparameter tuning on the Synthetic Minority Oversampling Technique (SMOTE) for addressing class imbalance issues in SDP. Using the Optuna framework for automated hyperparameter tuning, this study showed that optimized SMOTE (OpTunedSMOTE) outperformed standard SMOTE and other tuning methods. Notably, OpTunedSMOTE yielded an average increase of 20% in ROC-AUC and 18% in MCC scores across various classifiers. Random Forest and Extreme Gradient Boosting (XGB) classifiers showed the most significant improvements, with predictive performance gains of 22% and 21%, respectively, when combined with OpTunedSMOTE. However, the study also highlighted a trade-off, as hyperparameter tuning increased computational costs by an average of 30% in memory usage and execution time. These findings demonstrated that hyperparameter tuning of SMOTE significantly enhances model performance in handling class imbalance, a common issue in defect prediction.

Parameter optimization has proven highly effective for improving machine learning models, leading to enhanced predictive accuracy in software defect prediction (SDP). Building on these successes, we extended our approach to deep learning (DL) models by conducting a study focused on optimizing parameters specifically for DL-based SDP. This study examined the impact of hyperparameter tuning on Convolutional Neural Networks (CNNs), Long Short-Term Memory networks (LSTMs), Gated Recurrent Units (GRUs), and ensemble-based models for SDP. Using the Optuna framework, the study demonstrated substantial improvements in AUC scores for CNN and LSTM models, with a 20.87% and 19.75% increase, respectively, following hyperparameter tuning. GRUs showed moderate improvements with a 15.05% increase, while ensemble models displayed a high baseline performance and exhibited only a 6.74% increase in AUC post-tuning. Statistical analysis revealed that, after hyperparameter optimization, ensemble-based models emerged as the top performers, capitalizing on the strengths of multiple algorithms to achieve superior predictive accuracy and robustness. These findings underscored the importance of tailored hyperparameter tuning strategies, particularly for deep learning models, in enhancing defect prediction accuracy.

This thesis highlights the transformative impact of feature selection, machine learning, and deep learning techniques on enhancing Software Defect Prediction

(SDP) models. The findings can be concluded as follows:

1. Feature selection is essential for enhancing the accuracy and efficiency of Software Defect Prediction (SDP) models. By focusing on relevant features and removing redundant data, it reduces dimensionality, improving model performance.

2. Optimized feature selection methods, like the Grey Wolf Optimizer and Feature Subset Selection, significantly boost predictive accuracy, helping identify defect-prone areas early, enabling teams to allocate resources effectively, reduce costs, and improve software quality.

3. Both machine learning (ML) and deep learning (DL) classification techniques are key to SDP success. ML classifiers such as Random Forest and AdaBoost performed robustly, especially with parameter tuning, making them reliable choices for diverse datasets.

4. DL models, including CNNs and LSTMs, excelled with complex, high-dimensional data, achieving notable AUC improvements of 20.87% and 19.75% after tuning. DL ensembles emerged as top performers, showing enhanced accuracy and robustness.

5. Overall, ML offers effective, resource-efficient solutions, while DL, especially with optimized tuning, provides superior accuracy for complex datasets, positioning this research as a valuable contribution to scalable and reliable defect prediction models.

8.2 Application of the Work

The work conducted in the thesis would aid the software practitioners, researchers and society in the following ways:

1. **Aid to Researchers**

   This work provides significant insights and tools for researchers in software defect prediction, contributing to the advancement of the field through the following:

   - Establishes a structured framework for evaluating feature selection, extraction, and hyperparameter optimization techniques that researchers can apply or extend in their studies.
   - Offers detailed comparisons of both machine learning and deep learning classifiers, assisting researchers in identifying which methods work best for different dataset characteristics and prediction goals.
   - Introduces and validates novel optimization approaches, like the advanced version of Grey Wolf Optimizer, that researchers can adapt for other predictive modeling tasks within software quality assurance.

2. **Aid to Software Practitioners**

   This research provides practical strategies and tools for software developers and quality assurance teams, directly enhancing their work through:

   - Helps managers allocate resources effectively by identifying defect-prone areas early, allowing for better budget management and project timeline adherence.
   - Provides developers with optimized models for accurately detecting defect-prone code, enabling proactive quality improvements and reducing rework.

    - Enables testers to prioritize high-risk areas in the software, making testing efforts more efficient and improving defect detection accuracy.

3. **Aid to Society**

   The improvements in software defect prediction also benefit society by enhancing software reliability and usability, including:

    - Reduces the likelihood of software failures, leading to more robust, dependable applications that better meet users' needs and expectations.
    - Contributes to safer software in critical areas like healthcare, finance, and infrastructure, where reliable performance is essential for public safety.
    - Minimizes the costs of software maintenance and post-release fixes, supporting a more efficient and sustainable software development industry.

8.3 Future Work

While this thesis covers a wide range of feature selection techniques and classification models, future research could expand these experiments to datasets from different application domains and built using other programming languages, Doing so would increase the generalizability of the findings and broaden their applicability across various software ecosystems.

Further studies could also explore the application of advanced evolutionary algorithms to optimize feature selection techniques for Software Defect Prediction (SDP). Examining these algorithm's effectiveness for imbalanced datasets and predictive modelling would enhance the robustness of SDP methods.

Additionally, future research may benefit from hyperparameter tuning (HPT) of other preprocessing tasks, including data normalization, feature selection, and data balancing methods. Optimizing these preprocessing stages could lead to improved model accuracy and adaptability.

Replication remains essential, as it builds a stronger evidence base for researchers and practitioners, aiding in experimental planning and decision-making. It also enables the evaluation of results' applicability to diverse real-world scenarios or industrial settings. Therefore, future studies should aim to replicate these experiments to achieve more generalized and reliable conclusions.

Bibliography

[1] T. Menzies, J. DiStefano, A. Orrego, and R. Chapman, “Assessing predictors of software defects,” in *Proc. Workshop Predictive Software Models*, 2004, pp. 1–11.

[2] M. DâAmbros, M. Lanza, and R. Robbes, “Evaluating defect prediction approaches: a benchmark and an extensive comparison,” *Empirical Software Engineering*, vol. 17, pp. 531–577, 2012.

[3] 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.

[4] C. Tantithamthavorn, “Towards a better understanding of the impact of experimental components on defect prediction modelling,” in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 867–870.

[5] F. J. Buckley and R. Poston, “Software quality assurance,” *IEEE Transactions on Software Engineering*, no. 1, pp. 36–41, 1984.

[6] B. A. Kitchenham, “Software quality assurance,” *Microprocessors and microsystems*, vol. 13, no. 6, pp. 373–381, 1989.

[7] D. S. Alberts, “The economics of software quality assurance,” in *Proceedings of the June 7-10, 1976, national computer conference and exposition*, 1976, pp. 433–442.

[8] M. V. Mäntylä, F. Khomh, B. Adams, E. Engström, and K. Petersen, “On rapid releases and software testing,” in *2013 IEEE international conference on software maintenance*. IEEE, 2013, pp. 20–29.

[9] F. Akiyama, “An example of software system debugging.” in *IFIP congress (1)*, vol. 71, 1971, pp. 353–359.

[10] 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.

[11] A. E. Hassan, “Predicting faults using the complexity of code changes,” in *2009 IEEE 31st international conference on software engineering*. IEEE, 2009, pp. 78–88.

[12] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, “Predicting faults from cached history,” in *29th International Conference on Software Engineering (ICSE’07)*. IEEE, 2007, pp. 489–498.

[13] A. Mockus and D. M. Weiss, “Predicting risk of software changes,” *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[14] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 181–190.

[15] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, “Change bursts as defect predictors,” in *2010 IEEE 21st international symposium on software reliability engineering*. IEEE, 2010, pp. 309–318.

[16] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, “Bugcache for inspections: hit or miss?” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 322–331.

[17] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in *Third International Workshop on Predictor Models in Software Engineering (PROMISE’07: ICSE Workshops 2007)*. IEEE, 2007, pp. 9–9.

[18] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, “Software dependencies, work dependencies, and their impact on failures,” *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 864–878, 2009.

[19] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, “The impact of code review coverage and code review participation on software quality,” in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 192–201.

[20] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, “High-impact defects: a study of breakage and surprise defects,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 300–310.

[21] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, “Predicting the location and number of faults in large software systems,” *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[22] T. Zimmermann and N. Nagappan, "Predicting defects with program dependencies," in *2009 3rd international symposium on empirical software engineering and measurement*. IEEE, 2009, pp. 435–438.

[23] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 452–461.

[24] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 2007, pp. 364–373.

[25] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: an empirical case study," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 521–530.

[26] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 531–540.

[27] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 91–100.

[28] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? findings from a google case study," in

2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013, pp. 372–381.

[29] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[30] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 99–108.

[31] B. Caglayan, B. Turhan, A. Bener, M. Habayeb, A. Miransky, and E. Cialini, "Merits of organizational metrics in defect prediction: an industrial replication," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 89–98.

[32] J. Shimagaki, Y. Kamei, S. McIntosh, A. E. Hassan, and N. Ubayashi, "A study of the quality-impacting practices of modern code review at sony mobile," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 212–221.

[33] M. McDonald, R. Musson, and R. Smith, *The practical guide to defect prevention*. Microsoft Press, 2007.

[34] 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.

[35] 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.

[36] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE transactions on software engineering*, vol. 33, no. 1, pp. 2–13, 2006.

[37] I. Sommerville, "Software engineering 9th edition," *ISBN-10*, vol. 137035152, p. 18, 2011.

[38] M. Thomas and H. Thimbleby, "Computer bugs in hospitals: a new killer," *IT, Cybersecurity and Risk to Patients, Gresham College, Gresham College, available at:(accessed 26 February 2018)*, 2018.

[39] H. Wang, T. M. Khoshgoftaar, and A. Napolitano, "A comparative study of ensemble feature selection techniques for software defect prediction," in *2010 Ninth International Conference on Machine Learning and Applications*. IEEE, 2010, pp. 135–140.

[40] L. Qiao, X. Li, Q. Umer, and P. Guo, "Deep learning based software defect prediction," *Neurocomputing*, vol. 385, pp. 100–110, 2020.

[41] P. S. Bishnu and V. Bhattacherjee, "Software fault prediction using quad tree-based k-means clustering algorithm," *IEEE Transactions on knowledge and data engineering*, vol. 24, no. 6, pp. 1146–1150, 2011.

[42] J. M. Bieman, "Software metrics: A rigorous & practical approach," *IBM Systems Journal*, vol. 36, no. 4, p. 594, 1997.

[43] J. Nam, "Survey on software defect prediction," *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

[44] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.

[45] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 311–321.

[46] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, pp. 540–578, 2009.

[47] T. J. McCabe and C. W. Butler, "Design complexity measurement and testing," *Communications of the ACM*, vol. 32, no. 12, pp. 1415–1425, 1989.

[48] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[49] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 5, pp. 510–518, 1981.

[50] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 481–490.

[51] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 675–686, 2007.

[52] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: Recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 15–25.

[53] A. Bacchelli, M. D'Ambros, and M. Lanza, "Are popular classes more defect prone?" in *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, 2010, pp. 59–73.

[54] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *2010 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–10.

[55] A. B. Nassif, M. A. Talib, M. Azzeh, S. Alzaabi, R. Khanfar, R. Kharsa, and L. Angelis, "Software defect prediction using learning to rank approach," *Scientific Reports*, vol. 13, no. 18885, 2023.

[56] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016, pp. 33–45.

[57] Z. Li, J. Niu, and X.-Y. Jing, "Software defect prediction: future directions and challenges," *Automated Software Engineering*, vol. 31, no. 19, 2024.

[58] I. Arora, V. Tetarwal, and A. Saha, "Open issues in software defect prediction," *Procedia Computer Science*, vol. 46, pp. 906–912, 2015.

[59] J. Jiarpakdee, C. K. Tantithamthavorn, H. K. Dam, and J. Grundy, "An empirical study of model-agnostic techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 166–185, 2020.

[60] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano, "Rusboost: A hybrid approach to alleviating class imbalance," *IEEE transactions on systems, man, and cybernetics-part A: systems and humans*, vol. 40, no. 1, pp. 185–197, 2009.

[61] Z. Eivazpour and M. R. Keyvanpour, "Adversarial samples for improving performance of software defect prediction models," in *Data Science: From Research to Application*. Springer, 2020, pp. 299–310.

[62] P. S. Kumar and R. Venkatesan, "Improving software defect prediction using generative adversarial networks," *Int. J. Sci. Eng. Appl*, vol. 9, pp. 117–120, 2020.

[63] S. S. Rathore, S. S. Chouhan, D. K. Jain, and A. G. Vachhani, "Generative oversampling methods for handling imbalanced data in software fault prediction," *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 747–762, 2022.

[64] A. W. Dar and S. U. Farooq, "A survey of different approaches for the class imbalance problem in software defect prediction," *International Journal of Software Science and Computational Intelligence (IJSSCI)*, vol. 14, no. 1, pp. 1–26, 2022.

[65] S. K. Pandey and A. K. Tripathi, "Class imbalance issue in software defect prediction models by various machine learning techniques: an empirical study," in *2021 8th International Conference on Smart Computing and Communications (ICSCC)*. IEEE, 2021, pp. 58–63.

[66] D. Bassi and H. Singh, "A comparative study on hyperparameter optimization methods in software vulnerability prediction," in *2021 2nd International*

Conference on Computational Methods in Science & Technology (ICCMST). IEEE, 2021, pp. 181–184.

[67] R. Shu, T. Xia, J. Chen, L. Williams, and T. Menzies, “How to better distinguish security bug reports (using dual hyperparameter optimization),” *Empirical Software Engineering*, vol. 26, pp. 1–37, 2021.

[68] A. Agrawal, X. Yang, R. Agrawal, R. Yedida, X. Shen, and T. Menzies, “Simpler hyperparameter optimization for software analytics: Why, how, when?” *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2939–2954, 2021.

[69] 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.

[70] I. Tumar, Y. Hassouneh, H. Turabieh, and T. Thaher, “Enhanced binary moth flame optimization as a feature selection algorithm to predict software fault prediction,” *Ieee Access*, vol. 8, pp. 8041–8055, 2020.

[71] 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.

[72] N. S. Harzevili and S. H. Alizadeh, “Analysis and modeling conditional mutual dependency of metrics in software defect prediction using latent variables,” *Neurocomputing*, vol. 460, pp. 309–330, 2021.

[73] N. Gayatri, S. Nickolas, A. Reddy, S. Reddy, and A. Nickolas, “Feature selection using decision tree induction in class level metrics dataset for software defect predictions,” in *Proceedings of the world congress on engineering and computer science*, vol. 1, 2010, pp. 124–129.

[74] T. M. Khoshgoftaar and K. Gao, “Feature selection with imbalanced data for software defect prediction,” in *2009 International Conference on Machine Learning and Applications*. IEEE, 2009, pp. 235–240.

[75] Y. Xia, G. Yan, X. Jiang, and Y. Yang, “A new metrics selection method for software defect prediction,” in *2014 IEEE International Conference on Progress in Informatics and Computing*. IEEE, 2014, pp. 433–436.

[76] W. Han, C.-H. Lung, and S. Ajila, “Using source code and process metrics for defect prediction-a case study of three algorithms and dimensionality reduction.” *J. Softw.*, vol. 11, no. 9, pp. 883–902, 2016.

[77] L. Miao, M. Liu, and D. Zhang, “Cost-sensitive feature selection with application in software defect prediction,” in *Proceedings of the 21st international conference on pattern recognition (ICPR2012)*. IEEE, 2012, pp. 967–970.

[78] S. Goyal, “Genetic evolution-based feature selection for software defect prediction using svms,” *Journal of Circuits, Systems and Computers*, vol. 31, no. 11, p. 2250161, 2022.

[79] J. B. Awotunde, S. Misra, A. E. Adeniyi, M. K. Abiodun, M. Kaushik, and M. O. Lawrence, “A feature selection-based k-nn model for fast software defect prediction,” in *International Conference on Computational Science and Its Applications*. Springer, 2022, pp. 49–61.

[80] A. B. Nasser, W. Ghanem, A. S. H. Abdul-Qawy, M. A. Ali, A.-M. Saad, S. A. Ghaleb, and N. Alduais, “A robust tuned k-nearest neighbours classifier for software defect prediction,” in *International Conference on Emerging Technologies and Intelligent Systems*. Springer, 2022, pp. 181–193.

[81] X. Dong, Y. Liang, S. Miyamoto, and S. Yamaguchi, "Ensemble learning based software defect prediction," *Journal of Engineering Research*, vol. 11, no. 4, pp. 377–391, 2023.

[82] M. J. Hernández-Molinos, A. J. Sánchez-García, R. E. Barrientos-Martínez, J. C. Pérez-Arriaga, and J. O. Ocharán-Hernández, "Software defect prediction with bayesian approaches," *Mathematics*, vol. 11, no. 11, p. 2524, 2023.

[83] O. F. Arar and K. Ayan, "A feature dependent naive bayes approach and its application to the software defect prediction problem," *Applied Soft Computing*, vol. 59, pp. 197–209, 2017.

[84] M. A. Khan, N. S. Elmitwally, S. Abbas, S. Aftab, M. Ahmad, M. Fayaz, and F. Khan, "Software defect prediction using artificial neural networks: A systematic literature review," *Scientific Programming*, vol. 2022, no. 1, p. 2117339, 2022.

[85] A. Abdu, Z. Zhai, H. A. Abdo, R. Algabri, M. A. Al-Masni, M. S. Muhammad, and Y. H. Gu, "Semantic and traditional feature fusion for software defect prediction using hybrid deep learning model," *Scientific Reports*, vol. 14, no. 1, p. 14771, 2024.

[86] J. M. Catherine and S. Djodilatchoumy, "Multi-layer perceptron neural network with feature selection for software defect prediction," in *2021 2nd International Conference on Intelligent Engineering and Management (ICIEM)*. IEEE, 2021, pp. 228–232.

[87] Y. N. Soe, P. I. Santosa, and R. Hartanto, "Software defect prediction using random forest algorithm," in *2018 12th South East Asian Technical University Consortium (SEATUC)*, vol. 1. IEEE, 2018, pp. 1–5.

[88] S. Thapa, A. Alsadoon, P. Prasad, T. Al-Dalaâin, and T. A. Rashid, "Software defect prediction using atomic rule mining and random forest," in *2020 5th International Conference on Innovative Technologies in Intelligent Systems and Industrial Applications (CITISIA)*. IEEE, 2020, pp. 1–8.

[89] N. S. Thomas and S. Kaliraj, "An improved and optimized random forest based approach to predict the software faults," *SN Computer Science*, vol. 5, no. 5, p. 530, 2024.

[90] M. J. Siers and M. Z. Islam, "Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem," *Information Systems*, vol. 51, pp. 62–71, 2015.

[91] T. Sharma, A. Jatain, S. Bhaskar, and K. Pabreja, "Ensemble machine learning paradigms in software defect prediction," *Procedia Computer Science*, vol. 218, pp. 199–209, 2023.

[92] A. Alazba and H. Aljamaan, "Software defect prediction using stacking generalization of optimized tree-based ensembles," *Applied Sciences*, vol. 12, no. 9, p. 4577, 2022.

[93] B. Gezici and A. K. Tarhan, "Explainable ai for software defect prediction with gradient boosting classifier," in *2022 7th International Conference on Computer Science and Engineering (UBMK)*. IEEE, 2022, pp. 1–6.

[94] P. Bahad and P. Saxena, "Study of adaboost and gradient boosting algorithms for predictive analytics," in *International Conference on Intelligent Computing and Smart Communication 2019: Proceedings of ICSC 2019*. Springer, 2020, pp. 235–244.

[95] L. Lusa *et al.*, “Gradient boosting for high-dimensional prediction of rare events,” *Computational Statistics & Data Analysis*, vol. 113, pp. 19–37, 2017.

[96] D. Pradhan and D. Muduli, “Automated software defect prediction model: Adaboost-based support vector machine approach,” in *International Conference on VLSI, Signal Processing, Power Electronics, IoT, Communication and Embedded Systems*. Springer, 2023, pp. 257–270.

[97] ——, “Software defect prediction model using adaboost based random forest technique,” in *2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. IEEE, 2023, pp. 1–6.

[98] Y. Al-Smadi, M. Eshtay, A. Al-Qerem, S. Nashwan, O. Ouda, and A. Abd El-Aziz, “Reliable prediction of software defects using shapley interpretable machine learning models,” *Egyptian Informatics Journal*, vol. 24, no. 3, p. 100386, 2023.

[99] K. Wongpheng and P. Visutsak, “Software defect prediction using convolutional neural network,” in *2020 35th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*. IEEE, 2020, pp. 240–243.

[100] N. A. A. Khleel and K. Nehéz, “A novel approach for software defect prediction using cnn and gru based on smote tomek method,” *Journal of Intelligent Information Systems*, vol. 60, no. 3, pp. 673–707, 2023.

[101] J. Deng, L. Lu, and S. Qiu, “Software defect prediction via lstm,” *IET software*, vol. 14, no. 4, pp. 443–450, 2020.

[102] R. B. Bahaweres, D. Jumral, I. Hermadi, A. I. Suroso, and Y. Arkeman, “Hybrid software defect prediction based on lstm (long short term memory) and word

embedding," in *2021 2nd International Conference On Smart Cities, Automation & Intelligent Computing Systems (ICON-SONICS).* IEEE, 2021, pp. 70–75.

[103] H. Wang, W. Zhuang, and X. Zhang, "Software defect prediction based on gated hierarchical lstms," *IEEE Transactions on Reliability*, vol. 70, no. 2, pp. 711–727, 2021.

[104] H. S. Munir, S. Ren, M. Mustafa, C. N. Siddique, and S. Qayyum, "Attention based gru-lstm for software defect prediction," *Plos one*, vol. 16, no. 3, p. e0247444, 2021.

[105] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *Iet Software*, vol. 12, no. 3, pp. 161–175, 2018.

[106] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th international conference on predictive models in software engineering*, 2010, pp. 1–10.

[107] E. A. Felix and S. P. Lee, "Predicting the number of defects in a new software version," *PloS one*, vol. 15, no. 3, p. e0229131, 2020.

[108] W. Albattah and M. Alzahrani, "Software defect prediction based on machine learning and deep learning techniques: An empirical approach," *AI*, vol. 5, no. 4, pp. 1743–1758, 2024.

[109] N. Dhamayanthi and B. Lavanya, "Improvement in software defect prediction outcome using principal component analysis and ensemble machine learning algorithms," in *International Conference on Intelligent Data Communication Technologies and Internet of Things (ICICI) 2018.* Springer, 2019, pp. 397–406.

[110] S. K. Pandey, D. Rathee, and A. K. Tripathi, “Software defect prediction using k-pca and various kernel-based extreme learning machine: an empirical study,” *IET Software*, vol. 14, no. 7, pp. 768–782, 2020.

[111] Z. Xu, J. Liu, X. Luo, Z. Yang, Y. Zhang, P. Yuan, Y. Tang, and T. Zhang, “Software defect prediction based on kernel pca and weighted extreme learning machine,” *Information and Software Technology*, vol. 106, pp. 182–200, 2019.

[112] A. B. Nasser, W. A. H. Ghanem, A.-M. H. Saad, A. S. H. Abdul-Qawy, S. A. Ghaleb, N. A. M. Alduais, F. Din, and M. Ghetas, “Depth linear discrimination-oriented feature selection method based on adaptive sine cosine algorithm for software defect prediction,” *Expert Systems with Applications*, vol. 253, p. 124266, 2024.

[113] D. Bank, N. Koenigstein, and R. Giryes, “Autoencoders,” *Machine learning for data science handbook: data mining and knowledge discovery handbook*, pp. 353–374, 2023.

[114] L.-q. Chen, C. Wang, and S.-l. Song, “Software defect prediction based on nested-stacking and heterogeneous feature selection,” *Complex & Intelligent Systems*, vol. 8, no. 4, pp. 3333–3348, 2022.

[115] J. Dai and Q. Xu, “Attribute selection based on information gain ratio in fuzzy rough set theory with application to tumor classification,” *Applied Soft Computing*, vol. 13, no. 1, pp. 211–221, 2013.

[116] K. Gao and T. M. Khoshgoftaar, “Assessments of feature selection techniques with respect to data sampling for highly imbalanced software measurement data,” *International Journal of Reliability, Quality and Safety Engineering*, vol. 22, no. 02, p. 1550010, 2015.

[117] G. Sosa-Cabrera, M. García-Torres, S. Gómez-Guerrero, C. E. Schaerer, and F. Divina, "A multivariate approach to the symmetrical uncertainty measure: Application to feature selection problem," *Information Sciences*, vol. 494, pp. 1–20, 2019.

[118] G. Chandrashekar and F. Sahin, "A survey on feature selection methods," *Computers & electrical engineering*, vol. 40, no. 1, pp. 16–28, 2014.

[119] C. Lee and G. G. Lee, "Information gain and divergence-based feature selection for machine learning-based text categorization," *Information processing & management*, vol. 42, no. 1, pp. 155–165, 2006.

[120] Z. Wang, Y. Zhang, Z. Chen, H. Yang, Y. Sun, J. Kang, Y. Yang, and X. Liang, "Application of relieff algorithm to selecting feature sets for classification of high resolution remote sensing image," in *2016 IEEE international geoscience and remote sensing symposium (IGARSS)*. IEEE, 2016, pp. 755–758.

[121] P. Yang, W. Liu, B. B. Zhou, S. Chawla, and A. Y. Zomaya, "Ensemble-based wrapper methods for feature selection and class imbalance learning," in *Advances in Knowledge Discovery and Data Mining: 17th Pacific-Asia Conference, PAKDD 2013, Gold Coast, Australia, April 14-17, 2013, Proceedings, Part I 17*. Springer, 2013, pp. 544–555.

[122] A. O. Balogun, S. Basri, L. F. Capretz, S. Mahamad, A. A. Imam, M. A. Almomani, V. E. Adeyemo, A. K. Alazzawi, A. O. Bajeh, and G. Kumar, "Software defect prediction using wrapper feature selection based on dynamic re-ranking strategy," *Symmetry*, vol. 13, no. 11, p. 2166, 2021.

[123] M. A. Hall, "Correlation-based feature selection for machine learning," Ph.D. dissertation, The University of Waikato, 1999.

[124] K. Nagata, J. Kitazono, S. Nakajima, S. Eifuku, R. Tamura, and M. Okada, "An exhaustive search and stability of sparse estimation for feature selection problem," *IPSJ Online Transactions*, vol. 8, pp. 25–32, 2015.

[125] S. Nersisyan, V. Novosad, A. Galatenko, A. Sokolov, G. Bokov, A. Konovalov, D. Alekseev, and A. Tonevitsky, "Exhaufs: exhaustive search-based feature selection for classification and survival regression," *PeerJ*, vol. 10, p. e13200, 2022.

[126] F. Hussein, N. Kharma, and R. Ward, "Genetic algorithms for feature selection and weighting, a review and study," in *Proceedings of sixth international conference on document analysis and recognition*. IEEE, 2001, pp. 1240–1244.

[127] R. Sadeghi, R. Zarkami, K. Sabetraftar, and P. Van Damme, "Application of genetic algorithm and greedy stepwise to select input variables in classification tree models for the prediction of habitat requirements of azolla filiculoides (lam.) in anzali wetland, iran," *Ecological modelling*, vol. 251, pp. 44–53, 2013.

[128] A. K. Naik and V. Kuppili, "An embedded feature selection method based on generalized classifier neural network for cancer classification," *Computers in Biology and Medicine*, vol. 168, p. 107677, 2024.

[129] 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.

[130] A. O. Balogun, F. B. Lafenwa-Balogun, H. A. Mojeed, V. E. Adeyemo, O. N. Akande, A. G. Akintola, A. O. Bajeh, and F. E. Usman-Hamza, "Smote-based homogeneous ensemble methods for software defect prediction," in

Computational Science and Its Applications–ICCSA 2020: 20th International Conference, Cagliari, Italy, July 1–4, 2020, Proceedings, Part VI 20. Springer, 2020, pp. 615–631.

[131] P. D. Singh and A. Chug, “Software defect prediction analysis using machine learning algorithms,” in *2017 7th international conference on cloud computing, data science & engineering-confluence*. IEEE, 2017, pp. 775–781.

[132] R. Jayanthi and L. Florence, “Software defect prediction techniques using metrics based on neural network classifier,” *Cluster Computing*, vol. 22, pp. 77–88, 2019.

[133] D. Bowes, T. Hall, and J. Petrić, “Software defect prediction: do different classifiers find the same defects?” *Software Quality Journal*, vol. 26, pp. 525–552, 2018.

[134] 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.

[135] A. J. Bowers and X. Zhou, “Receiver operating characteristic (roc) area under the curve (auc): A diagnostic measure for evaluating the accuracy of predictors of education outcomes,” *Journal of Education for Students Placed at Risk (JESPAR)*, vol. 24, no. 1, pp. 20–46, 2019.

[136] J. Yao and M. Shepperd, “Assessing software defection prediction performance: Why using the matthews correlation coefficient matters,” in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, 2020, pp. 120–129.

[137] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE transactions on software engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[138] M. Friedman, "A comparison of alternative tests of significance for the problem of m rankings," *The annals of mathematical statistics*, vol. 11, no. 1, pp. 86–92, 1940.

[139] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics: Methodology and distribution*. Springer, 1992, pp. 196–202.

[140] J. Cohen, *Statistical power analysis for the behavioral sciences*. routledge, 2013.

[141] R. Malhotra, M. Khanna, and R. R. Raje, "On the application of search-based techniques for software engineering predictive modeling: A systematic review and future directions," *Swarm and Evolutionary Computation*, vol. 32, pp. 85–109, 2017.

[142] L. Kumar, S. Lal, A. Goyal, and N. B. Murthy, "Change-proneness of object-oriented software using combination of feature selection techniques and ensemble learning techniques," in *Proceedings of the 12th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*, 2019, pp. 1–11.

[143] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.

[144] S. S. Rathore and A. Gupta, "A comparative study of feature-ranking and

feature-subset selection techniques for improved fault prediction," in *Proceedings of the 7th India software engineering conference*, 2014, pp. 1–10.

[145] R. Malhotra and Y. Singh, "On the applicability of machine learning techniques for object oriented software fault prediction," *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 24–37, 2011.

[146] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering–a systematic literature review," *Information and software technology*, vol. 51, no. 1, pp. 7–15, 2009.

[147] Z. M. Hira and D. F. Gillies, "A review of feature selection and feature extraction methods applied on microarray data," *Advances in bioinformatics*, vol. 2015, no. 1, p. 198363, 2015.

[148] P. Jindal and D. Kumar, "A review on dimensionality reduction techniques," *Int. J. Comput. Appl*, vol. 173, no. 2, pp. 42–46, 2017.

[149] Z. Xu, J. Liu, Z. Yang, G. An, and X. Jia, "The impact of feature selection on defect prediction performance: An empirical comparison," in *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 2016, pp. 309–320.

[150] Z. A. Rana, M. M. Awais, and S. Shamail, "Impact of using information gain in software defect prediction models," in *International Conference on Intelligent Computing*. Springer, 2014, pp. 637–648.

[151] R. Malhotra and K. Khan, "A study on software defect prediction using feature extraction techniques," in *2020 8th International Conference on Reliability, In-*

focom Technologies and Optimization (Trends and Future Directions)(ICRITO). IEEE, 2020, pp. 1139–1144.

[152] S. A. Putri *et al.*, "Combining integrated sampling technique with feature selection for software defect prediction," in *2017 5th International Conference on Cyber and IT Service Management (CITSM)*. IEEE, 2017, pp. 1–6.

[153] 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.

[154] A. Saifudin, A. Trisetyarso, W. Suparta, C. Kang, B. Abbas, and Y. Heryadi, "Feature selection in cross-project software defect prediction," in *Journal of Physics: Conference Series*, vol. 1569, no. 2. IOP Publishing, 2020, p. 022001.

[155] S. Hosseini, B. Turhan, and M. Mäntylä, "A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction," *Information and Software Technology*, vol. 95, pp. 296–312, 2018.

[156] Q. Yu, S. Jiang, and J. Qian, "Which is more important for cross-project defect prediction: instance or feature?" in *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, 2016, pp. 90–95.

[157] J. Nam and S. Kim, "Heterogeneous defect prediction," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 508–519.

[158] M. Anbu and G. Anandha Mala, "Feature selection using firefly algorithm in software defect prediction," *Cluster Computing*, vol. 22, pp. 10 925–10 934, 2019.

[159] K. P. Ramulu and R. Murhtyr, “Importance of software quality models in software engineering,” *International Journal of Engineering Technologies and Management Research*, vol. 5, no. 3, pp. 200–218, 2018.

[160] M. Abdel-Basset, D. El-Shahat, I. El-Henawy, V. H. C. De Albuquerque, and S. Mirjalili, “A new fusion of grey wolf optimizer algorithm with a two-phase mutation for feature selection,” *Expert Systems with Applications*, vol. 139, p. 112824, 2020.

[161] S. Mirjalili, S. M. Mirjalili, and A. Lewis, “Grey wolf optimizer,” *Advances in engineering software*, vol. 69, pp. 46–61, 2014.

[162] M. Mustaqeem, S. Mustajab, and M. Alam, “A hybrid approach for optimizing software defect prediction using a grey wolf optimization and multilayer perceptron,” *International Journal of Intelligent Computing and Cybernetics*, vol. 17, no. 2, pp. 436–464, 2024.

[163] P. Dhavakumar and N. Gopalan, “An efficient parameter optimization of software reliability growth model by using chaotic grey wolf optimization algorithm,” *Journal of Ambient Intelligence and Humanized Computing*, vol. 12, pp. 3177–3188, 2021.

[164] S. Mallik, D. Pradhan, D. Muduli, A. Rath, G. Panda, S. Dash, and H. Qin, “A novel approach to enhance software defect prediction using an improved grey wolf optimization based extreme learning machine technique,” 2024.

[165] M. Prashanthi and M. Chandra Mohan, “Hybrid optimization-based neural network classifier for software defect prediction,” *International Journal of Image and Graphics*, vol. 24, no. 04, p. 2450045, 2024.

[166] R. Malhotra and K. Khan, “A novel software defect prediction model using two-phase grey wolf optimisation for feature selection,” *Cluster Computing*, pp. 1–23, 2024.

[167] I. M. El-Hasnony, S. I. Barakat, and R. R. Mostafa, “Optimized anfis model using hybrid metaheuristic algorithms for parkinsonâs disease prediction in iot environment,” *IEEE Access*, vol. 8, pp. 119 252–119 270, 2020.

[168] S. Mirjalili and A. Lewis, “S-shaped versus v-shaped transfer functions for binary particle swarm optimization,” *Swarm and Evolutionary Computation*, vol. 9, pp. 1–14, 2013.

[169] A. A. Heidari, S. Mirjalili, H. Faris, I. Aljarah, M. Mafarja, and H. Chen, “Harris hawks optimization: Algorithm and applications,” *Future generation computer systems*, vol. 97, pp. 849–872, 2019.

[170] Y. Khatri and S. K. Singh, “An effective feature selection based cross-project defect prediction model for software quality improvement,” *International Journal of System Assurance Engineering and Management*, vol. 14, no. Suppl 1, pp. 154–172, 2023.

[171] S. Mirjalili and A. Lewis, “The whale optimization algorithm,” *Advances in engineering software*, vol. 95, pp. 51–67, 2016.

[172] L. S. A. da Silva, Y. L. S. Lúcio, L. d. S. Coelho, V. C. Mariani, and R. V. Rao, “A comprehensive review on jaya optimization algorithm,” *Artificial Intelligence Review*, vol. 56, no. 5, pp. 4329–4361, 2023.

[173] T. Sharma and O. P. Sangwan, “Sine-cosine algorithm for software fault prediction,” in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 701–706.

[174] S. Riaz, A. Arshad, and L. Jiao, “Rough noise-filtered easy ensemble for software fault prediction,” *Ieee Access*, vol. 6, pp. 46 886–46 899, 2018.

[175] I. Gondra, “Applying machine learning to software fault-proneness prediction,” *Journal of Systems and Software*, vol. 81, no. 2, pp. 186–195, 2008.

[176] B. Ghotra, S. McIntosh, and A. E. Hassan, “Revisiting the impact of classification techniques on the performance of defect prediction models,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 789–800.

[177] 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, 2022.

[178] C. Seiffert, T. M. Khoshgoftaar, and J. Van Hulse, “Improving software-quality predictions with data sampling and boosting,” *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 39, no. 6, pp. 1283–1294, 2009.

[179] 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.

[180] D. Bassi and H. Singh, “The effect of dual hyperparameter optimization on software vulnerability prediction models,” *e-Informatica Software Engineering Journal*, vol. 17, no. 1, 2023.

[181] 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.

[182] N. Mittas and L. Angelis, "Ranking and clustering software cost estimation models through a multiple comparisons algorithm," *IEEE Transactions on software engineering*, vol. 39, no. 4, pp. 537–551, 2012.

[183] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 2623–2631.

[184] X. Qiu, P. Fan, and J. Ren, "Convolutional neural network-based research on software engineering defect prediction," in *Proceedings of the 6th International Conference on Information Technologies and Electrical Engineering*, 2023, pp. 305–308.

[185] G. Giray, K. E. Bennin, Ö. Köksal, Ö. Babur, and B. Tekinerdogan, "On the use of deep learning in software defect prediction," *Journal of Systems and Software*, vol. 195, p. 111537, 2023.

[186] I. Batool and T. A. Khan, "Software fault prediction using deep learning techniques," *Software Quality Journal*, vol. 31, no. 4, pp. 1241–1280, 2023.

[187] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

[188] K. Thirumoorthy *et al.*, "A feature selection model for software defect prediction using binary rao optimization algorithm," *Applied Soft Computing*, vol. 131, p. 109737, 2022.

[189] T. Mende and R. Koschke, “Revisiting the evaluation of defect prediction models,” in *Proceedings of the 5th international conference on predictor models in software engineering*, 2009, pp. 1–10.

Supervisor’s Biography

Prof. Ruchika Malhotra
Head of the Department & Professor
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)

Ruchika Malhotra is the Head of the Department and a Professor in the Department of Software Engineering at Delhi Technological University (DTU), Delhi, India. She has previously served as the Associate Dean of Industrial Research and Development at DTU. She was awarded the prestigious Raman Fellowship for post-doctoral research at Indiana University-Purdue University Indianapolis, USA. She earned her master’s and doctorate degrees in software engineering from the University School of Information Technology at Guru Gobind Singh Indraprastha University, Delhi, India. In 2013, she received the IBM Faculty Award. Her contributions to the field have earned her recognition as one of the worldâs top 2% scientists, according to a Stanford University report from 2020 to 2024, specifically for her work in ”Artificial Intelligence & Image Processing” . She has also received the Commendable Research Award from Delhi Technological University for the years 2018-2024. Her h-index is 38 as reported by Google Scholar. She is the author of the book ”Empirical Research in Software Engineering,” published by CRC Press, and co-author of ”Object Oriented Software Engineering,” published by PHI Learning. She has published over 250 research papers in international journals and conferences. Her research interests include software testing, software quality improvement, statistical and adaptive prediction models, software metrics, and the definition and validation of software metrics.

Author’s Biography

Kishwar Khan
Research Scholar
Department of Software Engineering
Delhi Technological University
Email: kishwarkhan037@gmail.com

Educational Qualifications:
M.Tech (SWE), B.Tech (CSE)

Kishwar Khan received her B.Tech. degree in Computer Science and Engineering from Dr. A. P. J. Abdul Kalam Technical University, Lucknow, India, and her M.Tech. degree in Software Engineering from Delhi Technological University, Delhi, India. She is currently pursuing a Ph.D. in Software Engineering at Delhi Technological University. Her doctoral research focuses on software quality improvement and the application of machine learning and deep learning in various aspects of software quality. Her current research interests include Artificial Intelligence, software quality, predictive modelling, and data analytics.

Cluster Computing
https://doi.org/10.1007/s10586-024-04599-w

A novel software defect prediction model using two-phase grey wolf optimisation for feature selection

Ruchika Malhotra[1] · Kishwar Khan[1]

Received: 8 March 2024 / Revised: 13 May 2024 / Accepted: 23 May 2024

Abstract
The process of accurately predicting software defects is highly crucial during the early period of software development before testing activities begin. A variety of computational methods have been constructed to achieve this based on static code metrics. However, one of the major issues in predictive modelling is the presence of redundant and irrelevant features in available datasets, which can lead to inaccuracies in the prediction model. Swarm optimization methods have shown excellent performance in Feature Selection (FS) issue mitigation and reduced the execution time of the prediction model. This study proposes a novel model for predicting software defects. This model utilizes a variant of Grey Wolf Optimiser as a wrapper-based feature selection method, paired with Synthetic Minority Oversampling Technique to balance the dataset, with the objective of maximizing the prediction efficiency of the learning model. The performance of the proposed model is assessed on 27 open-source datasets. The result findings show that the feature selection method improves prediction performance. Furthermore, the two-phase Grey Wolf Optimization-based feature selection with Random Forest classifier demonstrates superior efficacy on datasets compared to another benchmark model in handling the problem of FS. The results are also validated using statistical techniques.

Keywords Software defect prediction · Feature selection · Grey wolf optimisation · Synthetic minority oversampling technique (SMOTE) · Machine learning

1 Introduction

Software quality is a significant aspect of software engineering that needs to be considered when developing software products. The extent to which software conforms to customer needs and expectations is the software quality [1]. High-quality software products are more likely to meet user needs and expectations, resulting in greater user satisfaction. High-quality software is effective and efficient, with minimal defects. This can result in smoother business processes and better use of resources. Quality software development can reduce overall testing and maintenance efforts, resulting in lower costs. Reducing the number of faults through quality development practises might ultimately save money because correcting defects can be expensive. High-quality software can give organizations a competitive advantage in the marketplace. Customers are more likely to choose a software product that meets their needs and performs well. Standardizing software quality practices can improve overall software quality and promote consistency across organizations [2].

Additionally, by predicting software defects, software engineers can improve their overall development processes, as they can use the data generated by Software Defect Prediction (SDP) to identify parts of the code that are prone to defects and make changes to reduce the risk of future defects data [3]. SDP is the process of identifying those software components that are most likely to have defects based on current software metrics or features for prospective software releases [4]. Software characteristics may indicate software properties like complexity and the number of operators, and the connections between the features and the class may change. Some of these traits

✉ Kishwar Khan
kishwarkhan037@gmail.com

Ruchika Malhotra
ruchikamalhotra@dtu.ac.in

1 Department of Software Engineering, Delhi Technological University, Delhi, India

Gmail **Kishwar Khan <kishwarkhan037@gmail.com>**

Your Submission IDA-240485R1

Intelligent Data Analysis <em@editorialmanager.com> 23 August 2024 at 16:06
Reply-To: Intelligent Data Analysis <editor1@ida-ij.com>
To: Kishwar Khan <kishwarkhan037@gmail.com>

Ref.: Ms. No. IDA-240485R1
OpTunedSMOTE: A Novel Model for Automated Hyperparameter Tuning of SMOTE in Software Defect Prediction
Intelligent Data Analysis

Dear Ms. Khan,

I am pleased to tell you that your work has now been accepted for publication in Intelligent Data Analysis. It was accepted on Aug 21, 2024, comments from the Editor and Reviewers can be found below.

Your paper will now be sent to the publisher's typesetting office. Assuming there are no technical errors with the manuscript then, after the paper has been typeset, you will be informed by e-mail and asked to proofread the final copy. Please provide your feedback within 48 hours after receiving the proof.
After final publication in an issue of the journal you will receive a complimentary PDF copy of the final published article which will include the final page numbers.

Please complete the Author Publication Fee Payment form at the following link:
http://www.iospress.nl/journal-fee-form/?id=16&journal=29241.

More information is available at our website: http://www.iospress.nl/journal/intelligent-data-analysis/.

When you pay your publication fee, please make sure to enter your reference number (IDA-240485R1)

If you experience any problems with the payment, please contact Authorfees@iospress.nl

We would like to thank you again for all your efforts and your interest in the IDA journal, we look forward to receiving your future work as well.

With kind regards

IDA Journal

Jose M Pena, PhD
Editor-in-Chief
Intelligent Data Analysis

Comments from the Editors and Reviewers:

2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)
Amity University, Noida, India. June 4-5, 2020

A Study on Software Defect Prediction using Feature Extraction Techniques

Ruchika Malhotra
Department of Computer Science & Engineering
Delhi Technological University Shahbad Daulatpur,
Bawana Road, New Delhi, India
ruchikamalhotra@dtu.ac.in

Kishwar Khan
Department of Computer Science & Engineering
Delhi Technological University Shahbad Daulatpur,
Bawana Road, New Delhi, India
kishwarkhan037@gmail.com

Abstract— **Identification and elimination of defects in software is time and resource-consuming activity. The maintenance of a defective software system is burdensome. Software defect prediction (SDP) at an early stage of the Software Development Life Cycle (SDLC) results in quality software and reduces its development cost. In this study, a comparison is performed on nine open-source software-systems written in Java from PROMISE Repository using four mostly used feature extraction techniques such as Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA), Kernel-based Principal Component Analysis (K-PCA) and Autoencoders with Support Vector Machine (SVM) as base machine learning classifier. The model validation is performed using a ten-fold cross-validation method and the efficiency of the model is evaluated using accuracy and ROC-AUC. The results of this study indicate that Autoencoders is an effective method to reduce the dimensions of a software defect dataset successfully.**

Keywords— Software Defect Prediction, Feature Extraction, Dimensionality reduction, Machine Learning.

I. INTRODUCTION

Software defect prediction (SDP) is a process of classification that determines whether a software module is defective or not. It helps the Software Quality Assurance team to concentrate on the finite assets on the mostly defect susceptible software components. In this, every software component is categorized by a class tag and a number of metrics. The class tag indicates if a given module is defective or not [31].

Early prediction of defects may prompt to timely rectification of defects and leads to the delivery of maintainable software. Managers can allot testing resources suitably. Developers can audit defect-prone code more closely. Testers can prioritize their testing efforts and resources on the basis of defect-proneness data.

We know that machine learning methods are prevailing these days, various classification models have been presented during the previous decade. Despite that, a problem that scares the modeling method is the high-dimensionality of the dataset used for SDP, i.e., datasets with extreme features having irrelevant and redundant ones. As exposed by previous works, high-dimensionality issues can prompt high computational charge and deprivation of the accuracy of definite models [9], [10], [13]. Due to these reasons, a range of feature reduction techniques was projected to improve this problem of high dimensionality by removing extraneous & repeated features.

According to [30], with an increase in the dimensionality of data, there is an exponential increase in the amount of data needed to offer reliable performance, this fact is termed as the 'curse of dimensionality by Bellman when taking into account issues related to dynamic optimization. A well-liked undertaking to this issue of high-dimension datasets is to look for a projection of the data against a lesser amount of features, which conserve the information so far as possible. To conquer this issue, it is essential to discover a manner to reduce the number of variables in consideration [1], [8].

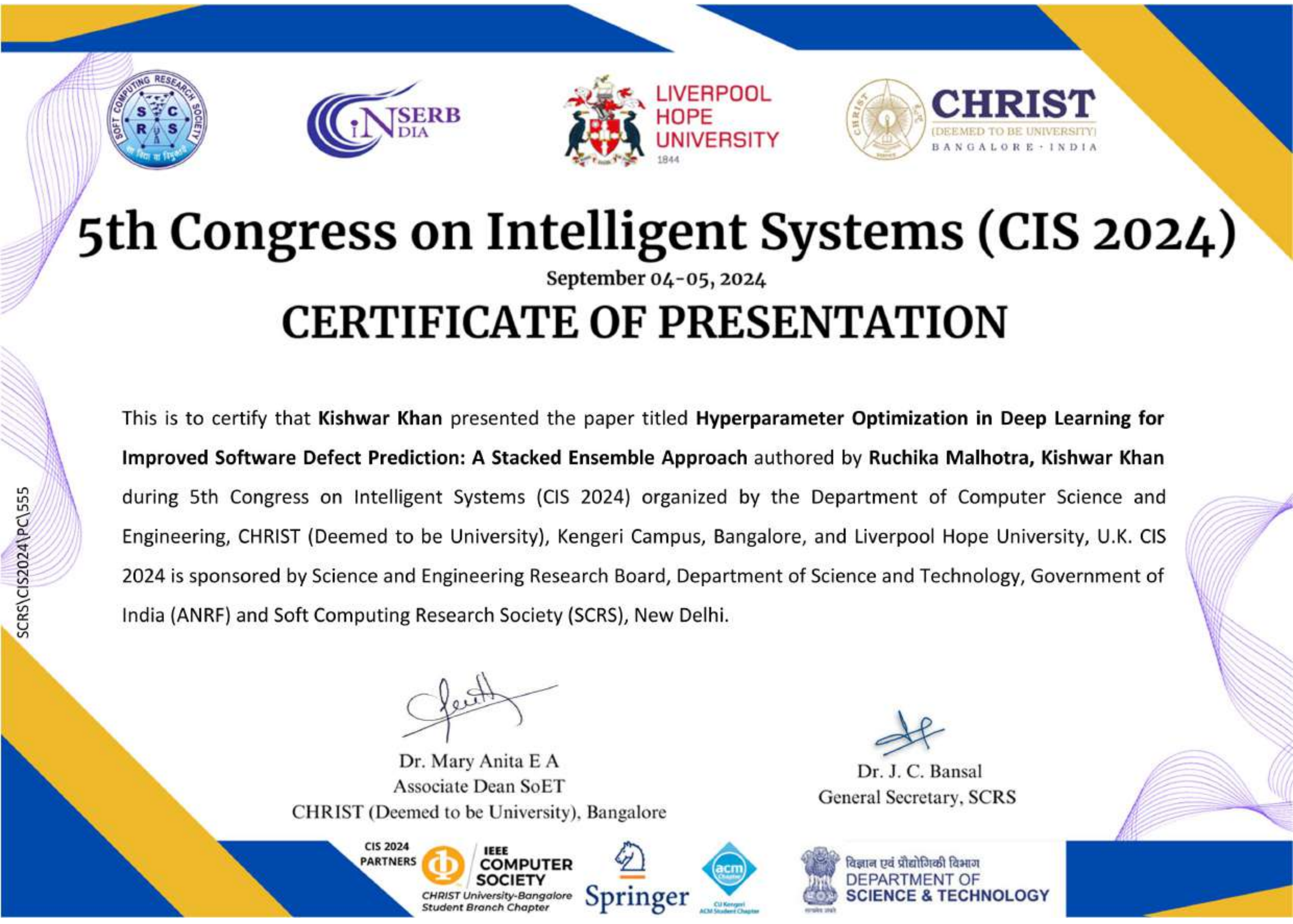
Feature extraction (FE) [24] is a technique to extract non-redundant and relevant features from the given set of features by using some transformation to decrease complications and to provide an easy demonstration of each variable in feature space as a linear arrangement of input variables. It is a more general technique than a feature selection technique. Principal Component Analysis, Linear Discriminant Analysis, and Kernel-based Principal Component Analysis are some of the approaches of feature extraction. In this study, we are comparing both linear as well as non-linear techniques.

The remaining part of the paper is arranged as follows: section II concisely presents the literature study and section III describes the research methodology used in this study. Section IV states the experimental design in which variable selection, hypothesis formulation, and tools required for the study are explained. Section V states and analyses the results of the study based on accuracy and ROC-AUC. Section VI concludes the study and provides guidelines for future work.

II. LITERATURE STUDY

Some of the previous studies about SDP are concisely summarized to depict the drift and trends in literature focusing on feature selection and extraction in SDP. Liu et al. [1] examined the effect of some 32 feature selection approaches such as filter-based, wrapper-based, clustering-based & extraction-based on the NASA dataset. Gayatri et al. [3] proposed a novel procedure for feature reduction build on Decision_Tree_Induction and compared it with the RELIEF method and discovered the proposed method outperformed others.

Ceylan et al. [4] conducted experiments in which PCA is used for feature-reduction. For classification Decision Tree, Multi-Layer Perceptron and Radial Basis Functions are used. Khoshgoftaar et al. [6] carried his research on the influence of data sampling, followed by wrapper-based feature selection. He found out that the proposed approach works



SCRS\CIS2024\PC\555

5th Congress on Intelligent Systems (CIS 2024)

September 04-05, 2024

CERTIFICATE OF PRESENTATION

This is to certify that **Kishwar Khan** presented the paper titled **Hyperparameter Optimization in Deep Learning for Improved Software Defect Prediction: A Stacked Ensemble Approach** authored by **Ruchika Malhotra, Kishwar Khan** during 5th Congress on Intelligent Systems (CIS 2024) organized by the Department of Computer Science and Engineering, CHRIST (Deemed to be University), Kengeri Campus, Bangalore, and Liverpool Hope University, U.K. CIS 2024 is sponsored by Science and Engineering Research Board, Department of Science and Technology, Government of India (ANRF) and Soft Computing Research Society (SCRS), New Delhi.

[illegible]

Dr. Mary Anita E A
Associate Dean SoET
CHRIST (Deemed to be University), Bangalore

[illegible]

Dr. J. C. Bansal
General Secretary, SCRS

CIS 2024 PARTNERS

IEEE COMPUTER SOCIETY CHRIST University-Bangalore Student Branch Chapter

Springer

CU Kengeri ACM Student Chapter

विज्ञान एवं प्रौद्योगिकी विभाग DEPARTMENT OF SCIENCE & TECHNOLOGY