Design and Implementation of a Software Bug Prediction Model Using Machine Learning Technique

MCS

Author

Hamza Khizar

Registration Number

00000320415

Supervisor

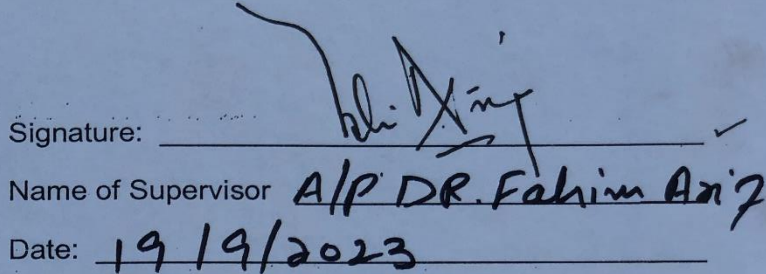
Assoc Prof Dr. Fahim Arif

A thesis submitted to the faculty of Department of Computer Software Engineering, Military College of Signals, National University of Sciences and Technology (NUST), Rawalpindi in partial fulfillment of the requirements for the degree of MS in Computer Software Engineering

(September 2023)

THESIS ACCEPTANCE CERTIFICATE

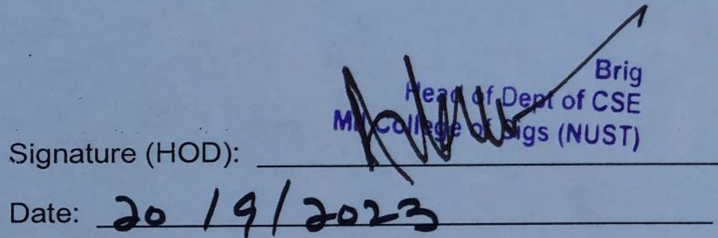
Certified that final copy of MS Thesis written by **Mr. Hamza Khizar**, Registration No. **00000320415**, of **Military College of Signals** has been vetted by undersigned, found complete in all respects as per NUST Statutes/Regulations/MS Policy, is free of plagiarism, errors, and mistakes and is accepted as partial fulfillment for award of MS degree. It is further certified that necessary amendments as pointed out by GEC members and local evaluators of the scholar have also been incorporated in the said thesis.



Signature: ______________________

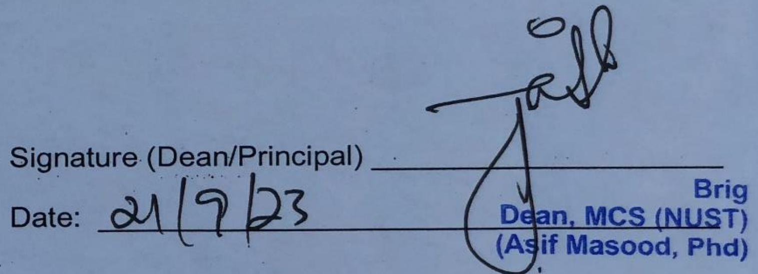
Name of Supervisor A/P DR. Fahim Aziz

Date: 19/9/2023



Signature (HOD): ______________________

Date: 20/9/2023



Signature (Dean/Principal) ______________________

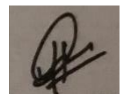
Date: 21/9/23

Declaration

I, *Hamza Khizar* declare that this thesis titled "Design and Implementation of a Software Bug Prediction Model Using Machine Learning Technique" and the work presented in it are my own and has been generated by me as a result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a Master of Science degree at NUST
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at NUST or any other institution, this has been clearly stated
3. Where I have consulted the published work of others, this is always clearly attributed
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work
5. I have acknowledged all main sources of help
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself



Hamza Khizar,
NUST00000320415 MSSE26

Dedication

I thank Almighty Allah, The Most Gracious and The Most Merciful.

This thesis is dedicated to my loving family, whose unwavering support and encouragement have been the cornerstone of my success. Their love and sacrifices have inspired me to strive for excellence in my studies and in all aspects of my life.

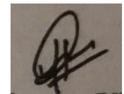
Abstract

Software Bug Prediction is an active research area and is being widely explored with the help of Machine Learning technique. The goal of bug prediction models is to identify potential software defects or bugs early in the development process, enabling developers to take preventive actions and improve software quality. Since bug prediction is now considered as an important measure of SDLC, there is need to have an efficient bug prediction model. Presently transfer learning, class imbalance and ensemble learning approaches are being researched much. In this research work an efficient model design is proposed and implemented. The proposed design caters the class imbalance issue of datasets as this is not much touched in the past. Class imbalance can affect the model accuracy by overfitting the model prediction results. The proposed design employ feature engineering technique which is used to add more domain information in the dataset for accurate prediction. Transfer learning is used to train and test the model on different datasets to analyze how much of the learning is passed to other dataset for cross project defect prediction; and ensemble method is utilized to explore the increase in performance upon combining multiple classifiers in a model. So, a model design is proposed which involve employing feature engineering, class imbalance and ensemble methods using machine learning technique for cross project defect prediction. Five NASA and four Promise datasets are used in the study for experimental analysis. Decision Tree (DT) and Random Forest (RF) are used as an individual base classifier. Three ensemble methods of bagging, boosting and stacking are used. The results shown that model attain the best accuracy with RF classifiers both as an individual and in

ensemble methods. The model has highest accuracy of 84% with RF as an individual classifier and also 84% with adaBoost in ensemble methods on NASA dataset. Whereas in PROMISE dataset, again RF have highest accuracy of 77% as an individual classifier and 79% with stacked ensemble method. Some other experiments are also conducted to evaluate buggy class recall score and it reveals that by using class imbalance, the recall of buggy class is high which indicates the model accuracy for prediction bugs in datasets.

Acknowledgment

Quite some time has passed since I began my degree, numerous things happened, various new places visited, survived a pandemic and life has changed altogether. However, it is time I finally close this chapter. I would like to thank my parents and my sisters for their constant support. A special thanks to my mother for keep asking me the thesis progress updates every day. I am indebted to my supervisor Dr. Fahim Arif for his politeness and firm belief in me and my committee members, Asst. Prof Dr. Yawar Abbas and Lt Col Khawir Mahmood, for always being there for guidance. Lastly, I want to thank my institution, MCS, for providing me the opportunity to learn and grow.



Hamza Khizar

Contents

Contents

Chapter 1 .... **1**
Introduction 1
1.1 Introduction 1
1.2 Purpose 1
1.3 Background 1
1.4 Software Bug Prediction 2
1.5 Scope 3
1.6 Problem Statement 4
1.7 Reason / Justification for the Selection of the Topic 5
1.8 Methodology 5
1.9 Objectives 5
1.10 Relevance to National Needs 6
1.11 Advantages 6
1.12 Area of Application 6
1.13 Thesis Outline 7
1.14 Summary 7
Chapter 2 .... **8**
Related Work 8
2.1 Introduction 8
2.2 Software Bug Prediction (SBP) 8
2.3 Machine Learning Techniques 9
2.3.1 Class Imbalance Handling 10
2.3.2 Cross Project Defect Prediction: 13
2.3.3 SBP Using Ensemble Methods 15
2.4 Summary 19
Chapter 3 .... **20**
PROPOSED DESIGN & METHDOLOGY 20
3.1 Introduction 20
3.2 Theoretical Concept of Proposed Design / Methodology 20
3.2.1 Transfer Learning 20
3.2.1.1 Benefits of using transfer learning in cross-project bug prediction 21
3.2.2 Ensemble Method 22
3.2.3 Class Imbalance 23
3.2.4 Feature Engineering 24
3.3 Proposed Model 26
3.3.1 Data Set: 27
3.3.1.1 NASA MDP Data Set 27
3.3.1.2 Promise Dataset 28
3.3.2 Data Preprocessing 29
3.3.3 Feature Engineering 30
3.3.4 Handling Data Class Imbalance 31
3.3.5 Machine Learning Classifier 31
3.3.5.1 Decision Tree 32
3.3.5.2 Random Forest 33
3.3.5.3 Bagging 33

3.3.5.4 AdaBoost 34
3.3.5.5 Stacking 34
3.3.6 Performance Metrics 35
3.3.6.1 Confusion Matrix 36
3.3.6.2 Accuracy 36
3.3.6.3 Precision 37
3.3.6.4 Recall 37
3.4 Summary 37
Chapter 4....................38
Implementation 38
4.1 Introduction 38
4.2 Environment (Hardware and Software) 38
4.3 Implementation Details 38
4.3.1 Data Pre-Processing (Step 1) 38
4.3.1.1 Null values and Noise 38
Implementation Steps 39
4.3.1.2 Dataset Scaling 39
Implementation Steps: 40
4.3.1.3 Handling Multiclass Nature of Promise Dataset: 40
Implementation Steps: 41
4.3.2 Feature Engineering (Step 2) 41
Implementation Steps: 41
4.3.3 Class Imbalance (Step 3) 42
Implementation Steps: 42
4.3.4 Dataset Division (Step 4) 43
4.3.5 Classification (Step 5) 44
4.4 Summary 48
Chapter 5....................**49**
Results and Discussion 49
5.1 Introduction 49
5.2 Implementation Details 49
5.3 Results 49
5.3.1 Base Classifiers Results for NASA Dataset 49
5.3.2 Ensemble Methods along with Base Classifiers Results for NASA Dataset 51
5.3.3 Base Classifiers Results for PROMISE Dataset 56
5.3.4 Ensemble Methods along with Base Classifiers Results for PROMISE Datas 56
5.3.5 Recall Comparison with and without Class Imbalance 60
5.3.6 Average Comparison Results on Both Repositories 61
5.4 Summary 63
Chapter 6....................63
Conclusion 64
6.1 Introduction 64
6.2 Conclusion 64
6.3 Future Work 65
6.4 Summary 65
References 66

List of Figures

Figure 2.1: Literature Review in the domain of Software Bug Prediction………………….. 9

Figure 3.1: Block Diagram of Proposed Model Design………………………………………….. 26

Figure 3.2: Detailed Diagram of Proposed Model Design……………………………………….. 27

Figure 3.3: Confusion Matrix………………………………………………………………………… 36

Figure 4.1: Multiclass Data of Ant Dataset………………………………………………………….. 41

Figure 4.2: Model Prediction Results Within Project…………………………….……...… 46

Figure 4.3: Model Prediction Results Across Projects……………………………...…… 47

Figure 5.1: Model Prediction Results on NASA Repository…………………………………….. 52

Figure 5.2: Model Prediction Results on PROMISE Repository………………………………... 58

Figure 5.3: Average Prediction Performance Comparison for NASA Repository……………... 62

Figure 5.4: Average Prediction Performance Comparison for PROMISE Repository………...63

List of Tables

Table 2.1: SBP Using Class Imbalance ………………………………………………...………11
Table 2.2: Cross-Project Bug Prediction………..………………………………….…..……...14
Table 2.3: SBP Using Ensemble Methods …………….……………………………….…..……...16
Table 3.1: Features in NASA Datasets……………………………………………………………28
Table 3.2: Features in PROMISE Datasets………………………………………………………29
Table 3.3: Cleaning Criteria of NASA Dataset…………………………………………...30
Table 3.4: Class Imbalance Handling……………………………………………………..31
Table 4.1: Class Imbalance Results on Both Data Repositories…………………………………43
Table 5.1: Prediction Performance of DT and RF on the NASA dataset………………………50
Table 5.2: Prediction Performance of Ensemble Methods on NASA Dataset …………………51
Table 5.3: Precision Score of Proposed Model on NASA Dataset………………………………53
Table 5.4: Recall Score of Proposed Model on NASA Dataset…………………………………..54
Table 5.5: F1-Score of Proposed Model on NASA Dataset………………………………………55
Table 5.6: Prediction Performance of DT and RF on PROMISE dataset………………………56
Table 5.7: Prediction Performance of Ensemble Methods on PROMISE datasets…………….57
Table 5.8: Precision Score of Proposed Model on PROMISE Dataset…………………………58
Table 5.9: Recall Score of Proposed Model on PROMISE Dataset…………………………….59
Table 5.10: F1-Score of Proposed Model on PROMISE Dataset………………………………..59
Table 5.11: Prediction Analysis With and Without Class Imbalance…………………………...60
Table 5.12: Average Prediction Performance Comparison for NASA Repository…………….61
Table 5.13: Average Prediction Performance Comparison for PROMISE Repository……….62

List of Abbreviations and Symbols

Abbreviations

SBP	Software Bug Prediction
SDP	Software Defect Prediction
CPDP	Cross-Project Defect Prediction
ML	Machine Learning
RF	Random Forest
KNN	K-Nearest Neighbors
NB	Naive Bayes
DT	Decision Tree
IT	Information Technology

Chapter 1

Introduction

1.1 Introduction

This chapter gives a general overview of the research work. It starts with an outline, purpose and background review of the proposed research work. Then it describes the motivation/ justification to carry out this work. The proposed methodology of the dissertation is discussed. The objectives, scope and national needs of the research work is also discussed. Finally, the sections of further chapters are defined.

1.2 Purpose

The goal of this research work is to use machine learning technique to design and implement an effective software bug prediction model. The focus is on to design a model by employing different parameters like feature engineering and class imbalance issue. Class imbalance affects the model's performance by over fitting the model and neglecting prediction of a buggy class. The designed model is implemented in Python. Two data repositories are used to access the accuracy of the proposed designed model using different software metrics.

1.3 Background

Each day brings new change in technology, especially in the world of Information Technology (IT). New software releases, new versions of old applications or languages come up, or entirely new techniques and programs innate. With such a rapid increase in software-based products, it has become a need to make these products successful and avoid any sort of failure. Defective software modules have

a significant impact on the quality of software, causing cost overruns, delays in delivery, and much higher maintenance expenses. Software Bug Prediction (SBP), which can directly affect the quality and has gained substantial popularity in recent years, is therefore the most crucial part of software.

A new dimension of mitigating the errors in software is being explored by the researchers and practitioners of industry. SBP methods are implemented to increase the efficiency, reduce failures, minimize resource loss and boost the performance. Machine learning approaches have shown to be quite successful in getting the desired outcomes for this purpose. Utilizing resources effectively for evaluating and testing programming modules is one of the key goals of defect prediction models. This helps in efficient allocation of resources in testing and fixing of error prone modules and hence producing high quality products at lower cost. Technically bug/defect predictor is a machine learning model applied on historical software metrics data to predict defects in software modules. The efficiency of model depends on quality of training data provided in addition to the machine learning classification technique used.

1.4 Software Bug Prediction

Software bug prediction refers to the process of identifying and predicting potential bugs or defects in software applications before they occur. This proactive approach allows developers and testers to focus their efforts on specific areas of the code that are more likely to have bugs, ultimately improving software quality and reducing the number of defects that make it to production.

There are various techniques and methods used for software bug prediction, including statistical analysis, machine learning, and data mining. These techniques often make use of historical data, such as previous bug reports, system logs, and code metrics, to identify patterns and indicators that can help predict the occurrence of bugs.

One common approach in bug prediction is through the use of metrics or code

complexity measures. By analyzing certain code metrics, such as lines of code, cyclomatic complexity, and code churn, developers can identify areas of the code that are more prone to defects. For example, if a specific module has a high cyclomatic complexity or has recently undergone many code changes, it may indicate a higher likelihood of bugs.

Machine learning algorithms are also widely used in bug prediction. These algorithms learn from historical bug data, code metrics, and other relevant information to build models that can predict the likelihood of bugs in different parts of the software. These models can then be used to prioritize testing efforts or allocate resources more efficiently.

Bug prediction can bring several benefits to software development. By identifying potential bugs early in the development process, developers can take preventive measures to address them, reducing the cost and effort required for bug fixing. It also helps in improving software reliability, reducing customer complaints, and enhancing user satisfaction. However, it is important to note that software bug prediction is not a foolproof solution. Predictive models are based on historical data and patterns, and they may not account for unforeseen factors or changes in the software development process. Therefore, bug prediction should be used as a complementary technique to traditional testing and not as a replacement.

In conclusion, software bug prediction is a proactive approach to identify and predict potential bugs in software applications. It leverages techniques like statistical analysis and machine learning to analyze historical data and code metrics to predict areas prone to defects. Bug prediction can help improve software quality, reduce bug fixing efforts, and enhance user satisfaction, but it should be used in conjunction with other testing methods.

1.5 Scope

The scope of software bug prediction is to identify and predict potential defects or bugs in software systems before they occur. This can include various aspects such

as code quality, design issues, configuration errors, or vulnerabilities. It aims to improve software quality and reduce the number of defects that occur in the production environment. By predicting bugs in advance, developers can take preventive measures to rectify the issues and avoid the negative consequences of bugs, such as system failures, security breaches, or customer dissatisfaction.

The scope of software bug prediction includes both static and dynamic analysis techniques. Static analysis involves examining the source code, design, or other artifacts without executing the program, whereas dynamic analysis involves monitoring the program during execution to identify issues.

Bug prediction can be applied throughout the software development life cycle, starting from the initial design phase to the testing and maintenance phases. It can also be used across different types of software projects, ranging from small-scale applications to complex, large-scale systems. Its scope can vary depending on the specific techniques and tools used. For example, some bug prediction approaches focus on code-level defects, while others may consider higher-level architectural issues or system-level vulnerabilities.

Overall, the scope of software bug prediction is to proactively identify and prevent defects in software systems, thereby improving software quality and reducing the potential negative impacts of bugs.

1.6 Problem Statement

The idea behind this research work is to develop a model or algorithm that can accurately predict the occurrence and location of bugs or defects in software applications. The goal is to identify potential issues early on in the development process, enabling developers to proactively address them and improve the overall software quality. This problem involves analyzing historical data, such as source code metrics, bug reports, and version control information, to detect patterns or indicators that can be used to predict the likelihood of future bugs. The challenge lies in finding the right set of features and developing a predictive model that can

generalize well to new software projects.

1.7 Reason / Justification for the Selection of the Topic

The literature review and industrial needs evidently show that an efficient model on this aspect of software development is warmly welcomed in market. Since the world is rapidly shifting towards software-based products, therefore our reliance of software has drastically increased. This gives rise to the idea of error-free software. A lot of models, principles and techniques are followed to achieve this notion such as small iterations, documentation, user interaction and well-organized process; still some inevitable defects occur causing a great distress to the software users and owners. Therefore, in order to mitigate these defects an efficient model for predicting them before they are born is necessary. This will give a boost in the performance of the final product and save much time and resources.

1.8 Methodology

The research aims to design an efficient model for software bug prediction and it involves 3 major steps: design of a model, implementation of a model and analysis of a model using different metrics. The design of a model involves data preprocessing, feature engineering, handling class imbalance and employing different machine learning classifiers for prediction. The process followed by implementing the same model in Python with individual classifiers and with ensemble methods also. The analysis includes prediction model accuracy, Precision, Recall and F1-score on two data repositories. The model detailed design and implementation details will be covered in Chapter 3 and 4 respectively.

1.9 Objectives

The main objectives of the study include

(i) Design of an efficient model for software bug prediction using machine learning techniques

(ii) Implementation of the same model in python language using datasets obtained from popular open-source software

(iii) Analysis of the implemented model using different performance measures for different databases

1.10 Relevance to National Needs

During the last decade, Pakistan has seen significant improvement in the software development with the sudden increase observed in software product usage globally. However, due to increased complexity, short time to market and high customer demands, often a software crisis or failure occurs which consume time and budget. This research will provide insights to predict the defects that become a critical challenge for system efficiency. The early defect prediction will help improve the software reliability and performance. This, in return, will give a major push to the national prosperity in IT world globally.

1.11 Advantages

It is necessary to discover the faults at early phase of software development to reduce the development cost and increase the success rate. This research will help to achieve this task in a well-structured way. Software testing, at the last stage of development cycle becomes painful with so many bugs coming up but with optimal prediction techniques, it will be made convenient and quicker.

1.12 Area of Application

This research will be utilized in the software industry, in detecting the bugs and their probable causes with the most efficient technique or model. All the software-based products will see a great spike in their performance and efficiency, be it medical software, safety critical software, e-commerce websites, home-based software products etc.

1.13 Thesis Outline

In the following chapters of thesis, different sections of research work are discussed at length.

Chapter 1 Introduction: An Overview of proposed research topic that includes introduction, motivation, scope, objectives and problem statement

Chapter 2 Related Work: Discussion and highlighting of work already carried out on this topic by other researchers.

Chapter 3 Proposed Design and Methodology: The design and methodology of the proposed Model is discussed in detail.

Chapter 4 Implementation: Explanation of the proposed model implementation in Python

Chapter 5 Results and Discussion: Model results on two data repositories are discussed using different software metrics

Chapter 6 Conclusion and Future Work: Final remarks about proposed model and future expansion is elaborated

1.14 Summary

In this chapter, introduction of research work is provided by giving background of the problem discussed. The scope and reason for choosing this research work is discussed. The proposed methodology, objectives, national needs, and advantages provide a bit more understanding about research work. At the end, area of application and organization of thesis is provided.

Chapter 2

Related Work

2.1 Introduction

In this chapter the already published work with relevance to software bug prediction, feature engineering and class imbalance is reviewed. It provides a general overview of the previous research work in this field.

2.2 Software Bug Prediction (SBP)

There are various approaches to create software bug prediction models mainly depending on factors like the required output, availability of datasets, features in a dataset, class imbalance handling and ML classifiers etc. The previous models often ignored some of the above-mentioned factors, which made them less effective. Later on, with the rapid growth of complexity of a software, the domain of software bug prediction became a popular research area in the field of software engineering. Many researchers are attracted towards this field proposing a variety of frameworks, models and techniques for bug prediction.

There are additional researchers who have concentrated on enhancing the currently utilized methods and models. Despite several efforts, there are still significant uncertainties in the field of software bug prediction research. Although numerous models and frameworks have been put forth, every method has its own drawbacks. To find bugs, several machine learning techniques are utilized, and datasets are made freely accessible so that practitioners can simply run their experiments without worrying about data. [1]

It is necessary to review the experimental data obtained on these techniques through the current studies in order to make machine learning techniques practicable in the context of bug prediction. [1] The figure 2.1 shows the elements what normally is

included in literature review in the area of SBP. Often, there are surveys or reviews conducted, discussions of previously used techniques, their pros and cons, the latest trends and famous topics as all this is very much required for a researcher to conduct a relevant and fruitful research project.

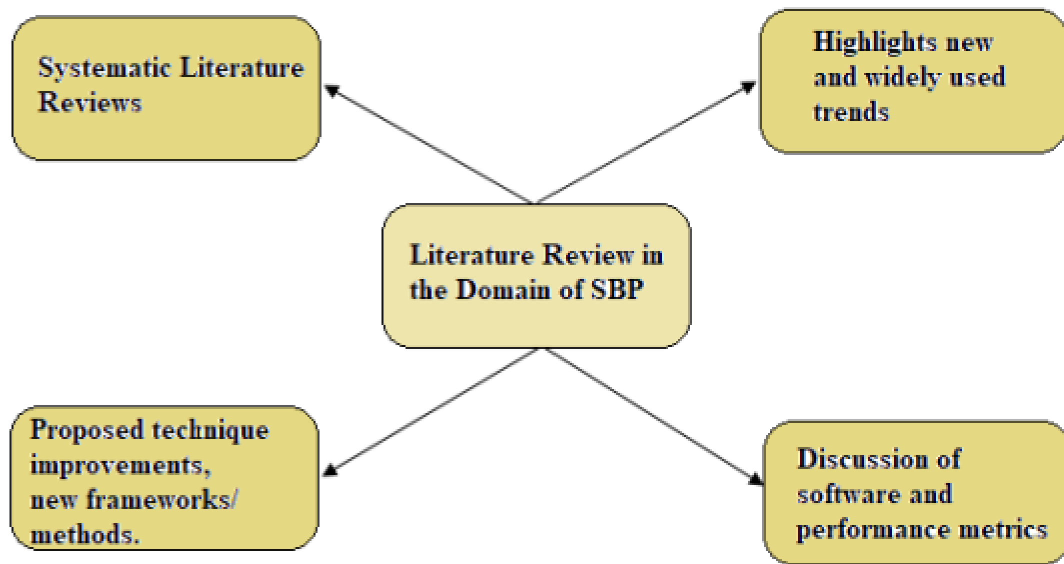


Figure 2.1: Literature Review in the domain of Software Bug Prediction

2.3 Machine Learning Techniques

From the literature review, it is found that various frameworks and techniques have been proposed to perform software bug prediction by combining data preprocessing, feature engineering, class imbalance and ensemble methods in a systematic manner to build models. Machine learning techniques, such as decision trees, random forests, and support vector machines, can be applied to predict software bugs. These algorithms learn from labeled datasets and develop models to classify whether a code segment is likely to contain a bug or not. Researchers have preferred them in order to ease the evaluation of the performance of their work. Apart from the mentioned approaches, different kinds of preprocessing methods are used in the cleaning of data and feature selection or feature ranking methods

are also utilized to reduce the dimensionality of dataset. After data cleaning, different classifiers are applied on the dataset, either individual classifiers or ensemble methods, to train the model.

2.3.1 Class Imbalance Handling

The problem of class imbalance has not been studied extensively in the last years. However, it impacts the model performance and an important factor which needs to be consider when designing a potential model. The following table 2.1 shows a list of related works using class imbalance for SBP. It shows the aim of study, ensemble method, dataset, performance measure and results.

Table 2.1: SBP Using Class Imbalance

Paper	Aim of the study	Used Techniques	Used Performance Measures	Results
[2]	Software defect prediction with class imbalance handling	Ensemble of decision trees (CSForest) and cost-sensitive voting (CSVoting)	Halstead metrics, cyclomatic complexity density, and weighted precision and recall	As a result, reported ensemble approaches outperformed decision tree classifiers, according to results.
[11]	Explore the use of ensemble learning for the software fault prediction	AdaBoost, Bagging, Random forest, Random subspace, and Voting	Precision, f-measure, and AUC	SMOTE method combined with random forest produced the greatest results.
[12]	Present a three stage ensemble learning framework for the software fault prediction 2018	Ensemble learning, feature selection, SMOTE, and Noise reduction techniques	Accuracy, f-measure, Root mean square error	A higher performance was obtained using the presented ELA technique with feature selection and data balancing. 95.4 with RaF DB
[14]	A comparison of ensemble methods for the software fault prediction 2018	Bagging, boosting, and rotation forest with 8 base learners	Accuracy and Recall	The best performance for software fault prediction, according to the results, was produced by rotation forest with resampling. 93.4 with rotation forest for KC1 DB.
[15]	Evaluation of sampling based ensembles on imbalanced data for software defect prediction	A ensemble method based on bagging mechanism and SMOTE	Accuracy, Precision, Recall and f1-score	According to the findings, sampling approaches and ensemble learning improved prediction performance, when combined together.
[17]	Cross-project software defect prediction (CPDP)	A cost-sensitive boosting approach	Probability of detection, probability of false alarm, G-mean, and Balance	The results demonstrated that the CPDP performed better when transfer learning and class imbalance were combined.
[18]	Cross-project software defect prediction 2019	AdaBoost, Bagging, and SMOTE	Accuracy, True positive rate, False positive rate, and AUC	Results showed that the SMOTE worked best when paired with the bagging technique. 73%
[21]	Use of a coding-based ensemble learning for the software defect	Class imbalance technique, three coding-based ensemble methods	AUC and Ranking	The approaches that were presented outperformed four additional base classifiers (random forest, C4.5, naive Bayes,

	prediction			and Ripper), bagging, boosting, and cost-sensitive learning.
[23]	A comparative study of ensemble-based learning for the software fault prediction 2017	Bagging and AdaBoost.M1 with J48 and DT as base learners	Accuracy and AUC	The outcomes demonstrated that ensemble learning, feature selection, and data balance increased performance. 97.3 using the MC1 database and SMOTEIGDTAdaBoost.M1
[28]	Handling of imbalanced data using ensemble learning in software defect prediction	AdaBoost.Ml, AdaBoost.M2, RUSBoost, SMOTEBoost, MSMOTEBoost, DataBoost	AUC, G-mean, and Balance	RUSBoost, MSMOTEBoost, and SMOTEBoost ensemble approaches generated the best performance, according to the results.
[31]	Handling software defect prediction for imbalance data 2016	AdaBoost and back-propagation neural network (BPNN)	Accuracy, Precision, Recall, Specificity, F-measure, and G-mean	According to the results, Adaboost performed better than BPNN for the imbalanced fault datasets. 91.3 with PC4 database and BP-Adaboost
[37]	Cross-project software defect prediction (CPDP)	Value-cognitive boosting and support vector machine (VCB-SVM)	AUC, H-measure, Probability of detection and Probability of false alarm	In comparison to the current CPDP approach and the current class imbalance techniques, the VCB-SVM offered superior prediction performance.
[53]	SMOTE and homogeneous ensemble (Bagging and Boosting) methods are proposed for predicting software defects.		Implemented using Bayesian networks and decision trees as the primary classifiers.	The proposed technique achieved high accuracy of 86.8% and an area under the operating receiver characteristics curve value of 0.93%.

2.3.2 Cross Project Defect Prediction:

Cross-project defect prediction refers to predicting defects in a target project based on data from other projects. This approach is used when the target project has limited or no historical defect data available, but there is sufficient data from other similar projects. The idea is to transfer the knowledge and patterns learned from the source projects to the target project.

Cross-project defect prediction uses machine learning techniques to build a predictive model using data from multiple projects. It involves identifying relevant features (e.g., code complexity, developer experience, etc.) and training a model on historical defect data from different projects. This model is then applied to the target project to predict the likelihood of defects in the code.

There are various challenges in cross-project defect prediction, including the differences in software characteristics, coding practices, and development environments across projects. These differences can affect the accuracy and effectiveness of the predictive model. Therefore, it is important to carefully select appropriate source projects that have similarities with the target project and consider the transferability of the learned models.

Cross-project defect prediction can be beneficial for projects that have limited defect data, especially for early defect detection and prioritizing limited testing resources. However, it should be used as a supplementary approach and not solely rely on the predictions, as the transferability and generalizability of the models may vary across projects. The following table 2.2 shows a list of related works performing SBP for cross-project.

Table 2.2: Cross-Project Bug Prediction

Paper	Aim of the study	Used Technique	Used Performance Measures	Results
[17]	Cross-project software defect prediction (CPDP)	A cost-sensitive boosting approach	Probability of detection, probability of false alarm, G-mean, and Balance	The results demonstrated that the CPDP performed better when transfer learning and class imbalance were combined.
[18]	Cross-project software defect prediction 2019	AdaBoost, Bagging, and SMOTE	Accuracy, True positive rate, False positive rate, and AUC	Results showed that the SMOTE worked best when paired with the bagging technique. 73%
[33]	Analysis of meta-heuristic algorithm and ensemble methods for the cross-project defect prediction	Harmony search-based optimization and cost-sensitive boosting	Probability of detection, probability of false alarm, G-means, and Balance	The employment of cost-sensitive boosting and meta-heuristic algorithms in combination with parameter tuning improved the performance of the strategies, according to the results.
[37]	Cross-project software defect prediction (CPDP)	Value-cognitive boosting and support vector machine (VCB-SVM)	AUC, H-measure, Probability of detection and Probability of false alarm	In comparison to the current CPDP approach and the current class imbalance techniques, the VCB-SVM offered superior prediction performance.
[38]	Cross-project software defect prediction (CPDP)	A cost-sensitive boosting approach (TCSBoost)	Probability of detection, Probability of false alarm, G-mean, and Balance	The CPDP performance was enhanced as a result of the suggested TCSBoost.
[40]	Use of combined classifier for cross-project defect prediction (CPDP)	Meta classification algorithm (CODEPLogistic) and 5 base learners	F-measure, Precision, Recall, Mean average precision, and cost effectiveness	Results showed that CODEPLogistic was surpassed by Bootstrap aggregation with J48, which produced the best results.
[42]	Ensemble of regression approaches for the cross-project fault prediction	Ensemble of regression	Area under the ROC curve	Results demonstrated that the proposed method outperformed regression-based approaches.
[45]	Heterogeneous defect prediction	Krnel spectral embedding transfer ensemble (KSETE)	Probability of detection, AUC, G-measure, and MCC	The outcomes of the experiment shown that KSETE is efficient in both HDP and CPDP-CM scenarios.

2.3.3 SBP Using Ensemble Methods

Ensemble methods in software bug prediction refer to the combination of multiple prediction models or algorithms to improve the accuracy and reliability of defect prediction. These methods combine the predictions from different models in various ways, such as bagging, boosting, stacking, averaging, voting, or weighting, to make a final prediction. These methods have been shown to improve prediction performance compared to single models. They benefit from the diversity of the individual models and leverage their strengths to make more accurate and robust predictions. The following table 2.3 shows a list of related works using ensemble methods for SBP.

Table 2.3: SBP Using Ensemble Methods

Paper	Aim of the study	Used Ensemble Techniques	Used Performance Measures	Results
[3]	Analysis of combining feature selection and ensemble learning for the software defect classification	Forward selection and average probability ensemble (APE) with SVM	G-means and area under ROC curve (AUC)	Even with weak features, the presented APE approach worked as intended, and when combined with forward selection, the AUC values were increased.
[4]	Explore the use of deep learning and two-stage ensemble (TSE) for the software defect prediction	Deep learning and stacking with TSE (SDAEsTSE)	F-measure, AUC, and MCC	When compared to Random Forest, Bagging, and AdaBoost, the performance of the presented SDAEsTSE technique was much greater.
[5]	Hybridize ensemble methods for the just-in-time defect prediction	Two-layer ensemble learning (TLEL) with decision tree and ensemble learning	Cost-effectiveness and F1-measure	Results indicated that TLEL only used 20% of the code and found over 70% of the errors. The F1-scores of TLEL were noticeably higher than those of other approaches.
[7]	Software defect prediction using ensemble techniques 2018	Weighted majority voting (WM), Randomized weighted majority voting (RWM), Cascading weighted majority voting (CWM), and Cascading randomized weighted majority voting (CRWM) techniques.	Accuracy, f-measure, and AUC	Metrics that take change into consideration perform superior to other metrics. Used ensembles performed better than basic classifiers. precision of 96.6%
[8]	Software fault prediction at various metrics levels using ensemble methods	Bagging, boosting, stacking, and voting	Root mean square error, and AUC	Bagging outperformed other ensemble techniques for the datasets used.
[9]	Evaluation of ensemble methods for software fault prediction 2015	AdaboostM1, Vote and StackingC with Naïve Bayes, Logistic, J48, VotedPerceptron, and SMO as base learners	Accuracy and f-measure	StackingC outperformed baseline learners and other applied ensemble techniques. 97.76%
[10]	Ensemble based software defect prediction using diversity selection	Weighted accuracy diversity (WAD) and stacking	Precision, MCC, and diversity	Naïve Bayes and SMO provided better results with the presented WAD technique.
[13]	Software defect prediction using heterogeneous ensemble method	A hybrid ensemble approach using different classifiers	Precision, Recall, and G-means	For all scenarios taken into consideration, the presented ensemble methods beat bagging, Adaboost, and other base learning techniques.

[16]	Software fault prediction in the large space systems	Apriori, Decision tree, K-nearest neighbor, naïve Bayes, Support vector machine, a majority voting based ensemble method	Error rate	According to the results, ensembles using a decision tree classifier and the Naive Bayes classifier both had higher accuracy rates.
[19]	Ensemble learning for the software defect prediction	Multiple kernel ensemble learning (MKEL) technique	F-measure, probability of detection, probability of false alarm	The presented MKEL strategy outperformed previous methods regarded as state-of-the-art and gave better outcomes for all 12 datasets used.
[20]	Prediction of software black-box defects	Stacked generalization approach (PMoSG)	Root mean square error and correlation coefficient	The PMoSG technique outperformed LibSVM, LR, IBK, Decision trees, and MLP in terms of prediction performance.
[22]	Exploration of ensemble methods for the fault prediction in the Eclipse projects	Bagging, Boosting, Random Subspace, Rotation Forest, and stacking	Average absolute error and average relative error	Random subspace outperformed other used ensemble methods. The performance of rotation forest, bagging, and boosting was average.
[24]	Use of linear homogeneous ensemble for software fault prediction	Extreme Learning Machine Ensemble (DELME) and Non-differentiable Extreme Learning Machine Ensemble (NELME)	Average absolute error, average relative error, MoC, Prediction at level 1	Proposed methods performed consistently better for all the used datasets.
[25]	Static and dynamic ensemble approaches for the software fault prediction	Over-Bagging and ensemble selection, Omni-Ensemble Learning (OEL)	Probability of detection, probability of false alarm, AUC, and G-mean	For all of the performance metrics that were employed, OEL produced the best results.
[26]	Analysis of heterogeneous ensembles for online failure prediction	Decision tree, bagging, random forest, gradient boosting, and stacking with soft voting	Confusion matrix parameters	The results showed that mixing weak learners frequently enhanced the accuracy of predictions.
[29]	Software defect prediction with cost-sensitive boosting approach	Cost-sensitive boosting neural networks (CSBNN)	Misclassification rate, type I and type II errors	Among the models that were used, CSBNN consistently had the lowest error rate and had good performance.
[30]	A source code metric and ensemble approach for the software fault prediction 2017	Best Training Ensemble, Majority Voting Ensemble, Nonlinear Ensemble Decision Tree Forest with five base learners	Accuracy, F-measure, and cost-analysis	The findings demonstrated that majority voting ensemble gave the best outcomes and that ensemble-based prediction models are also economically advantageous.

[34]	Explore the use of ensemble techniques for the bug assignment in the large industrial contexts 2016	Stacked Generalization (SG) with five base learners	Accuracy	SG consistently performed better than basic students. Results also indicated that in order to properly train SG, bug assignment must require at least 2,000 bug reports. 89%
[35]	Heterogeneous software defect prediction	Two-Stage Ensemble Learning (TSEL), Ensemble Multiple Kernel Correlation Alignment (EMKCA), and RESample with replacement (RES) techniques	AUC, Precision, Recall, F-measure, and Balance	The outcomes demonstrated that TSEL outperformed the standard approaches.
[36]	An industrial case study of ensemble methods for locating software defects	Neural network, naïve Bayes, and voting feature intervals	Probability of false alarm (PF), Probability of detection (PD), and Balance	The proposed ensemble method combined a voting strategy to decrease PF and boost precision.
[39]	A hybrid approach for the software defect prediction 2011	Analytic Hierarchy Process (AHP) and ensemble methods (bagging, boosting, and stacking)	Accuracy, TPR, FPR, TNR, FNR, Precision, Recall, F-measure, AUC, Kappa, and MAE	The best results are produced by AdaBoost, and the top base learner was K-nearest Neighbor. 92.63
[41]	A comparison of ensemble techniques with feature selection for the software fault prediction	17 different feature ranking techniques and their ensemble using stacking and Naïve Bayes algorithm	F-measure, Odd-ratio, Probability ratio, AUC, G-means, and Area under precision and recall curve	Results indicated that no ensemble method in particular outperformed other applied methods. Additionally, ensembles of a few rankers performed better than those of many or all rankers.
[46]	Classifier ensemble-based software defect prediction	Bagging, Boosting, Random trees, Random Forest, Random subspace, Stacking, and Voting	Area under the ROC curve	Results indicated that random forest and majority voting produced the best results. .
[48]	Heterogeneous ensemble methods for the software fault prediction	Four ensemble methods (ERWT, LRWT, DTF, and GBR) with three base learners, DTR, MLP, and LR	Average absolute, Average relative error, Prediction at level 1, and measure of completeness	The findings demonstrated that non-linear rule-based methods (DTF and GBR) performed the fault prediction the best.

2.4 Summary

This chapter sheds light on the previous work done in the domain of software bug prediction, the approaches, methods, advantages and limitations of their work is shown. The widely used dataset, tools and approaches for creating a prediction model has also been considered.

PROPOSED DESIGN & METHDOLOGY

3.1 Introduction

This chapter gives details overview of proposed design, its theoretical concept and methodology. The design of the proposed research project is discussed in detail and the methodology is also described. The analysis of the used data set, their features, relevancy of features and metrics are also discussed. The machine learning classifier, tools, and performance metrics used in this work are also explained. This chapter concludes with design and methodology of efficient bug prediction model using machine learning technique that will help to predict software bugs with greater accuracy.

3.2 Theoretical Concept of Proposed Design / Methodology

The proposed research project involves a design and implementation of an efficient model for software bug prediction. It requires clean dataset with relevant features, efficient classifier and valid training and testing of model for efficient bug prediction. For this purpose, distinct methods are used to build the model more robust and novel among the previously built models discussed in the literature survey. These methods are discussed as follows:

- **Transfer Learning**
- **Ensemble Method**
- **Class Imbalance**
- **Feature Engineering**

3.2.1 Transfer Learning

Software bug prediction can be performed within the same project or on cross-

projects. Transfer learning is used in cross-project software bug prediction and it refers to the use of pre-trained models on one project and predicts bugs in another project. This approach recognizes that software projects often share common characteristics and patterns, even across different domains or applications. The typical process of transfer learning in cross-project bug prediction involves the following steps:

- **Pre-training**: A model is trained on a source project(s) that contains labeled bug data. The model learns to understand the underlying patterns and features associated with bug-prone code.
- **Knowledge Transfer**: After pre-training, the trained model's knowledge is transferred to a different target project that has limited labeled bug data. This step involves adapting the pre-trained model to the target project by fine-tuning or retraining it on the available labeled bug data in the target project.
- **Prediction**: Once the transfer learning process is complete, the adapted model is used to predict bugs in the target project. The model leverages the knowledge gained from the source project to make accurate predictions on the bug-proneness of the target project's code.

3.2.1.1 Benefits of using transfer learning in cross-project bug prediction

Limited labeled data requirement: Using knowledge transferred from a source project, the target project can benefit from the pre-trained model's understanding of bugs without requiring a massive amount of labeled bug data.

Improved bug prediction accuracy: The pre-trained model has already learned general patterns and features associated with bugs, making it more likely to make accurate predictions on the target project's code.

Time and resource efficiency: Transfer learning allows for the reuse of pre-trained models, reducing the time and resources needed to train a model from scratch for

each project. This approach can speed up the bug prediction process for new projects.

While transfer learning offers promising advantages, it is crucial to consider the differences between the source and target projects such as codebase, programming language, or domain. The transferability of knowledge depends on the similarity and level of commonality between the projects. Therefore, careful analysis and adaptation of the pre-trained model to the target project are necessary to ensure effective bug prediction.

In this research work, the cross-project defect prediction is performed where source and target data are from different projects. The model is built using one project considered as source project and employed for prediction on another project called as target project. The features in both the projects are kept same but feature engineering technique is employed for model efficient training and prediction.

3.2.2 Ensemble Method

Machine learning techniques called ensemble methods combine several models or algorithms to increase overall performance and prediction accuracy. The underlying idea is that by combining weak models, a stronger and more accurate model is obtained. There are several popular ensemble methods, including:

Bagging: This method involves training multiple models on different subsets of the training data and combining their predictions through majority voting or averaging. The most commonly used algorithm for bagging is the Random Forest, which combines multiple decision trees.

Boosting: In boosting, models are trained sequentially, where each subsequent model focuses on the instances that the previous models struggled to predict accurately. The final prediction is made by combining the outputs of all models. Gradient Boosting (XG) and Adaptive Boosting (AdaBoost) are popular boosting algorithms.

Stacking: In stacking, multiple models are trained and their forecasts are combined

using a meta-learner or meta-model, a different model.

The meta-learner learns to combine the predictions of the base models, potentially achieving better performance.

Voting: Voting methods combine the predictions of multiple models by majority voting or weighted averaging. There are different types of voting, such as hard voting (majority voting) and soft voting (weighted averaging based on predicted probabilities).

Ensemble methods provide better predictive performance than using a single model alone in many cases. They help in reducing overfitting, improving generalization, and handling bias-variance trade-off. Ensemble approaches, however, could need more training data than a single model and can be computationally expensive. In this research work Bagging, Ada boosting and stacking are employed for bug prediction.

3.2.3 Class Imbalance

Class imbalance occurs when the number of instances in one class is much lower than the number of instances in another. In the context of software bug prediction, this means that the number of instances representing bugs (the minority class) is much smaller than the number of instances representing non-bugs (the majority class).

Class imbalance can pose challenges in software bug prediction because traditional machine learning algorithms tend to favor the majority class, leading to biased and inaccurate predictions. This is because these algorithms are typically designed to minimize overall error, which leads them to focus on the majority class and ignore the minority class. As a result, the model may have poor performance in predicting the minority class (bugs).

To address the issue of class imbalance in software bug predictions, several techniques can be employed:

Oversampling the minority class: To boost the minority class's representation in

the training data, this entails creating synthetic instances of the minority class. Techniques like SMOTE (Synthetic Minority Over-sampling Technique) can be used to create synthetic instances based on the existing minority class instances.

Undersampling the majority class: This involves lowering the number of instances in the majority class to match the number of instances in the minority class. This can be done by randomly removing instances or using more advanced techniques like Tomek Links or Cluster Centroids.

Using ensemble methods: Ensemble methods combine multiple models to make predictions. They can be effective in dealing with class imbalance by combining models that are trained on different subsets of the data or using techniques like boosting, where the focus is shifted towards the minority class.

Adjusting class weights: Many machine learning algorithms provide the option to assign different weights to different classes. By assigning higher weights to the minority class, the algorithm gives it more importance during training and evaluation.

Changing the evaluation metric: Instead of using traditional metrics like accuracy, precision, recall, or F1-score, evaluation metrics specific to imbalanced datasets, such as Area Under the Precision-Recall Curve (AUPRC) or Cohen's kappa coefficient, can be used to assess the model's performance more accurately.

It is significant to remember that the technique selected relies on the particular situation and dataset. Different techniques may work better in different scenarios, so experimentation and evaluation are crucial to finding the most effective approach. In this research work, oversampling technique is employed to handle the class imbalance in the datasets.

3.2.4 Feature Engineering

Feature engineering is a crucial step in building software bug prediction models. It involves selecting, transforming, and creating relevant features from raw data that can effectively represent the characteristics of software systems and help improve

the performance of bug prediction models. Some common feature engineering techniques used in software bug prediction includes:

Metrics-based features: Software systems often generate various metrics, such as code complexity, code churn, and code ownership. These metrics can be used as features to capture important aspects of software systems that may impact bug occurrence. For example, the number of code changes or the number of developers modifying a particular module can be important indicators of bug-prone areas.

Textual features: Bug reports, source code comments, and documentation can be valuable sources of information for feature engineering. Techniques like text mining, natural language processing (NLP), and information retrieval can be used to extract useful features from these textual data. For example, keywords related to software modules, error messages, or specific bug-fixing activities can be important indicators of bug-prone areas.

Temporal features: Considering the temporal aspect of software development can be useful in bug prediction. Features such as the number of bugs reported in the past, the time since the last bug fix, or the number of code changes over time can provide insights into the dynamics of the software system and potentially help identify bug-prone periods.

Social features: In collaborative software development environments, features related to social interactions among developers can be informative. For example, features like the number of code reviews, code ownership distribution, or developer network centrality can capture the social dynamics of the development process and potentially influence bug occurrence.

Code structure and dependencies: Features related to the software code structure and dependencies can also be valuable. These features might include the size of code modules, code coupling and cohesion measures, or architectural properties of the system. Such features can provide insights into the structural complexity and organization of the software, which can impact bug occurrence.

Domain-specific features: Depending on the specific software domain, additional

features related to the application domain can be considered in feature engineering. For example, if the software is dealing with financial data, features related to financial metrics or risk indicators can be incorporated.

It is important to note that effective feature engineering requires domain knowledge and a deep understanding of the software development process. Iterative refinement and experimentation with different feature combinations are often necessary to achieve optimal performance in bug prediction models.

3.3 Proposed Model

The model design is proposed to carry out efficient prediction of bugs in a software. In the proposed design, transfer learning, feature engineering, class imbalance and ensemble methods are used for efficient bug prediction. The block diagram of the proposed design is shown in Figure 3.1 which has mainly five parts i.e., labeled data availability, handling class imbalance, feature engineering, creating training set and building prediction model. The detailed diagram of the same is shown in Figure 3.2.

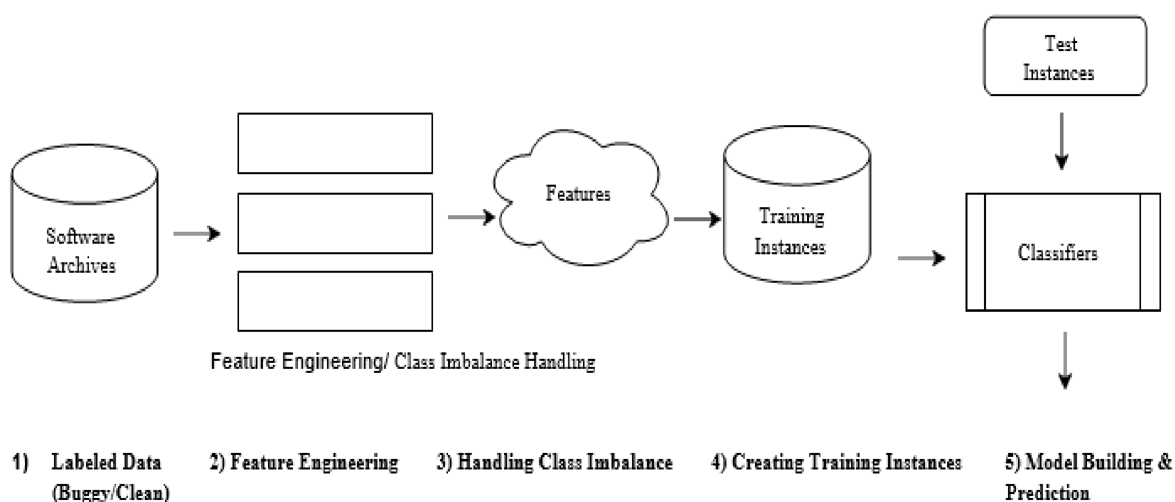


Figure 3.1: Block Diagram of Proposed Model Design

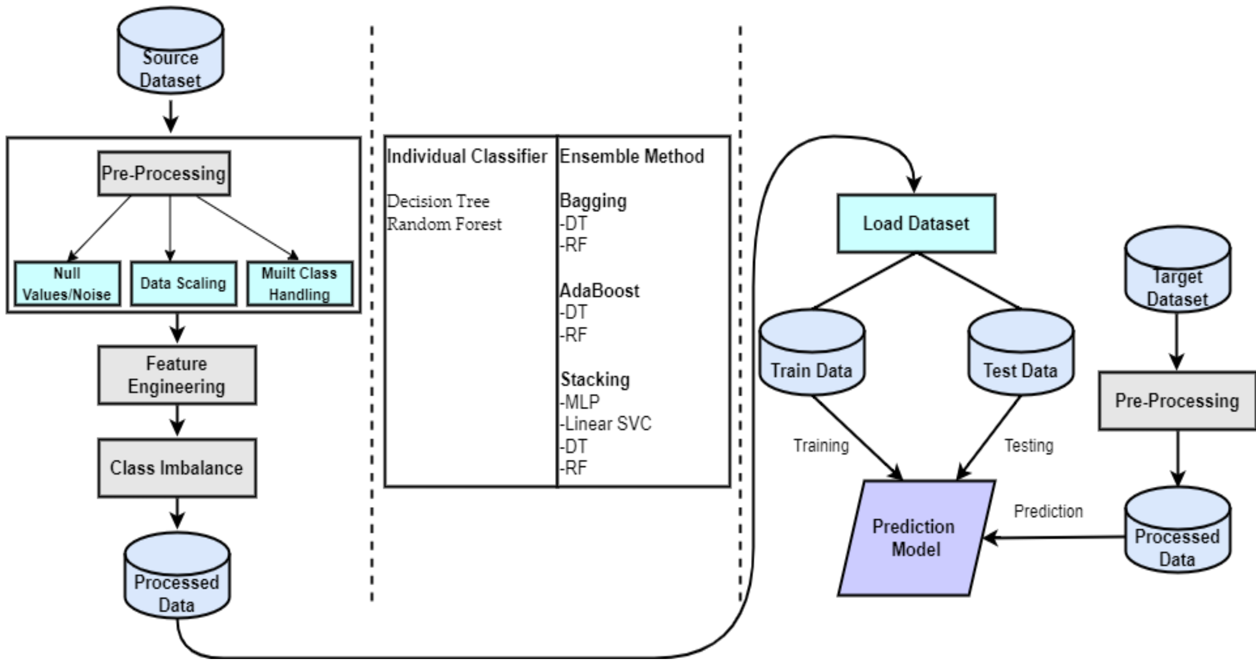


Figure 3.2: Detailed Diagram of Proposed Model Design

3.3.1 Data Set:

The effectiveness of the suggested model is assessed on five NASA benchmark datasets and four datasets from Promise Repository. These datasets are publicly available and consist of historical data of software modules. Many studies have utilized these datasets in their research and this is the primary reason of our interest in them as it will be easier to compare our results with them. The selected datasets include several features and a known output class that determines the defectiveness of an instance. Based on data available for other features, this output class is predicted by the prediction model. The datasets have many projects with various attributes, sizes, and defective rates that help to check the generality of research. [55]

3.3.1.1 NASA MDP Data Set

From the NASA MDP Dataset's CM1, MW1, PC1, PC3, and PC4 subsets are selected, which are made available to the public on the PROMISE Software Engineering

Repository. Table 3.1 shows the features present in NASA datasets. McCabe and Halstead source code extractors provide data from software for storage management for receiving and analyzing ground data. These characteristics were defined in the 1970s in an attempt to objectively characterize code characteristics related with software quality. [56]

Table 3.1: Features in NASA Datasets

ID	Feature Name	ID	Feature Name
1.	LOC_BLANK	20.	HALSTEAD_EFFORT
2.	BRANCH_COUNT	21.	HALSTEAD_ERROR_EST
3.	CALL_PAIRS	22.	HALSTEAD_LENGTH
4.	LOC_CODE_AND_COMMENT	23.	HALSTEAD_LEVEL
5.	LOC_COMMENTS	24.	HALSTEAD_PROG_TIME
6.	CONDITION_COUNT	25.	HALSTEAD_VOLUME
7.	CYCLOMATIC_COMPLEXITY	26.	MAINTENANCE_SEVERITY
8.	CYCLOMATIC_DENSITY	27.	MODIFIED_CONDITION_COUNT
9.	DECISION_COUNT	28.	MULTIPLE_CONDITION_COUNT
10.	DECISION_DENSITY	29.	NODE_COUNT
11.	DESIGN_COMPLEXITY	30.	NORMAL_CYCLOMATIC_COMPLEXITY
12.	DESIGN_DENSITY	31.	NUM_OPERANDS
13.	EDGE_COUNT	32.	NUM_OPERATORS
14.	ESSENTIAL_COMPLEXITY	33.	NUM_UNIQUE_OPERANDS
15.	ESSENTIAL_DENSITY	34.	NUM_UNIQUE_OPERATORS
16.	LOC_EXECUTABLE	35.	NUMBER_OF_LINES
17.	PARAMETER_COUNT	36.	PERCENT_COMMENTS
18.	HALSTEAD_CONTENT	37.	LOC_TOTAL

3.3.1.2 Promise Dataset

The data in Promise dataset refers to open-source Java systems and ant-1.7, camel-1.6, ivy-2.0 and xalan-2.4 are selected for experiments in this research work. The features present in them are shown in table 3.2. The table shows all of the twenty features present in the dataset. The first column displays the feature ID while the second and third column shows the feature name and detail respectively. These IDs are used in another table to show the selected features which are used in this study.

Table 3.2: Features in PROMISE Datasets

ID	FEATURE NAME	FEATURE DETAIL
1.	wmc	Weighted methods per class
2.	dit	Depth of inheritance tree
3.	noc	Number of children
4.	cbo	Coupling between object classes
5.	rfc	Response for a class
6.	lcom	Lack of cohesion in methods
7.	ca	Afferent couplings
8.	ce	Efferent couplings
9.	npm	Number of public methods
10.	lcom3	Lack of cohesion in methods, different from LCOM
11.	loc	Lines of code
12.	dam	Data access metric
13.	moa	Measure of aggregation
14.	mfa	Measure of functional abstraction
15.	cam	Cohesion among methods of class
16.	ic	Inheritance coupling
17.	cbm	Coupling between methods
18.	amc	Average method complexity
19.	max_cc	Maximum McCabe's cyclomatic complexity
20.	avg_cc	Average McCabe's cyclomatic complexity

3.3.2 Data Preprocessing

First step in the proposed design after dataset selection is data preprocessing. Two version of NASA datasets are provided by [57]. DS' refers to version of dataset that includes duplicate and inconsistent instances whereas DS" refers to dataset that does not include redundant and inconsistent instances. Originally, these datasets were available at NASA website; however, they are removed from this source. Backup of 12 cleaned NASA datasets is available at [58]. 5 cleaned and widely used

datasets are selected from the available datasets available at [58] which include CM1, MW1, PC1, PC3, PC4. Previous studies have already discussed and used these cleaned versions of datasets in their experiments. The other four datasets have been taken from PROMISE repository available at [59]. They contain 20 Object Oriented metrics as independent features and defect-proneness of class as dependent variable. The criteria of cleaning as stated in [57] is shown in Table 3.3.

Table 3.3: Cleaning Criteria of NASA Dataset

Sr. No	Category of Data Quality	Description
1.	Identical cases	In this case multiple instances have same values for all features including class label
2.	Inconsistent cases	This is the situation where two or more instances have same values for all features except for class label.
3.	Cases with missing values	This case refers to instances that contain one or more missing observations.
4.	Cases with conflicting feature values	In this situation, an instance has two or more metric values that violate some referential integrity constraint. For example, LOC TOTAL is less than COMMENTED LOC. However, COMMENTED LOC is a subset of LOC TOTAL.
5.	Cases with implausible values	This case refers to instances that violate some integrity constraint. For example, value of LOC=1.1

3.3.3 Feature Engineering

In the proposed design, Domain-specific features engineering is explored. Depending on the existing five features of software domain which includes BRANCH_COUNT, CONDITION_COUNT, CYCLOMATIC_COMPLEXITY, DECISION_COUNT and NUMBER_OF_LINES are used to create an additional feature and added in the model training phase. This feature helps model to learn more efficiently that at a particular threshold the features have specific value of being buggy or not.

3.3.4 Handling Data Class Imbalance

The literature review suggest that the problem of Class imbalance had not been studied in detail. However, it has an important role in model prediction capability. Keeping this in view, the class imbalance has been incorporated in the proposed design. There are many techniques as discussed above to cater class imbalance, but in the proposed design, Oversampling has been used which **Oversample the minority class**. It involves generating instances of the minority class to increase its representation in the training data. The notion of "Yes" indicates that there is a bug in data instance and "No" indicates that there is no bug in the data instance. The class information of all data sets before and after class imbalance have been shown below in table 3.4.

Table 3.4: Class Imbalance Handling

Sr.No.	**Dataset**	**Dataset Class Info before Class Imbalance**		**Dataset Class Info After Class Imbalance**	
		Yes	**No**	**Yes**	**No**
1	**CM1**	42	285	285	285
2	**MW1**	27	226	226	226
3	**PC1**	61	644	644	644
4	**PC3**	134	943	943	943
5	**PC4**	177	1110	1110	1110
6	**Ant-1.7**	166	579	579	579
7	**Camel-1.6**	188	777	777	777
8	**Ivy-2.0**	40	312	312	312
9	**Xalan-2.4**	110	613	613	613

3.3.5 Machine Learning Classifier

A model is the result of the classifier's machine learning, whereas a classifier is an algorithm or collection of rules used to categorize or classify data. The model is trained using the classifier, and the classifier is then used by the model to categorize the data. In the scenario of this study, this step consists of choosing individual

classifiers that were mostly used in artificial intelligence and the integration of well-known algorithm to form an ensemble-learning model. In the first step, I chose two individual classifiers i.e., Decision Tree and Random Forest. These are frequently used classifiers and give good performance in defect prediction. [60,61] In the second step, ensemble-learning method is proposed where the trained base classifier in the first step, used ensemble classifiers to create a model. For ensemble methods Bagging, AdaBoost and Stacking are used. The source dataset after passing through preprocessing, feature selection and class imbalance phase was trained using these individual classifiers and then with ensemble method. Trained model was then tested on source dataset to see which classifier achieved better accuracy values. On the other hand, target dataset is also preprocessed and then by using the same trained and tested model, prediction is performed on target dataset. The Individual and ensemble classifiers used in this study are descried below:

3.3.5.1 Decision Tree

A graphical representation of a series of decisions is called a decision tree, that lead to a particular outcome. It is a way of visualizing and understanding the decision-making process. Every node in the tree indicates a decision or a test on a particular feature, and each branch represents an outcome or a possible result of that decision or test. The tree starts with a root node and ends with leaf nodes, which represent the final outcomes.

Decision trees are commonly used in various fields such as data mining, machine learning, and business analytics. They are particularly useful when dealing with classification or regression problems, where the objective is to forecast or estimate a target variable based on a set of input variables or features.

Some advantages of using decision trees include their simplicity and interpretability. Decision trees are easy to understand and visualize, making them useful for explaining the logic and reasoning behind a particular decision. They can also handle both categorical and numerical data, making them suitable for a wide

range of problems.

3.3.5.2 Random Forest

An ensemble learning technique for classification and regression applications. It is a type of supervised learning algorithm that combines multiple decision trees to make predictions.

In Random Forest, multiple decision trees are created using a subset of the training data and a random subset of the input features. Each decision tree is trained independently on the different subsets of data to generate a prediction. During prediction, the random forest algorithm takes the majority vote from all the decision trees to make the final prediction.

Random Forests have several advantages over a single decision tree. They reduce overfitting by creating multiple decision trees and combining their predictions. They can handle a large number of input features and are able to capture non-linear relationships between features and the output variable. Random Forests are also capable of handling missing values and outliers in the data.

Random Forests have various applications and are commonly used in fields like finance, healthcare, and e-commerce. They can be used for predicting stock prices, diagnosing diseases, and recommending products to users, among other tasks.

3.3.5.3 Bagging

Bagging, also known as bootstrap aggregating, is a technique used in machine learning for improving the accuracy and stability of models. It involves creating multiple subsets of the original dataset through random sampling with replacement, training a separate model on each subset, and then combining their predictions through averaging or voting.

Bagging is commonly used with decision trees, where each model in the ensemble is trained on a different random sample of the training dataset. As each model may have different strengths and weaknesses, the combination of their predictions can

lead to better overall performance.

Bagging helps to reduce overfitting by inducing diversity among the models, as each model is trained on a slightly different subset of the data. It also helps in reducing bias by reducing the variance of the models' predictions.

Bagging can be used for both regression and classification tasks. In the case of regression, the final prediction is usually the average of the predictions from each model. In classification, the final prediction can be determined by majority voting or by taking the class with the highest probability.

Overall, bagging is a powerful technique for improving the accuracy and robustness of machine learning models, especially when dealing with complex and noisy datasets.

3.3.5.4 AdaBoost

AdaBoost is an ensemble learning method that combines multiple weak classifiers to create a strong classifier. It works by sequentially training weak classifiers on different subsets of the training data. In each iteration, the algorithm gives more weight to misclassified samples, so the subsequent weak classifiers focus on correctly classifying these samples. The final classifier is a weighted combination of the weak classifiers, where the weights are determined based on their individual performance.

Advantages of AdaBoost:

- AdaBoost is a versatile algorithm that can be used with various base classifiers.
- It is less prone to overfitting, as the algorithm focuses on misclassified examples in each iteration.
- It performs well in practice and has been widely used in various domains of machine learning.

3.3.5.5 Stacking

Stacking is an ensemble learning method that combines multiple classifiers or regression models to improve the overall performance. It involves training multiple

models, also known as base models or learners, on a given dataset. These base models can be of any type, such as decision trees, support vector machines, or neural networks.

The stacking ensemble method consists of two stages:

1. **Base Models Training:** In this stage, various base models are trained using the input data. Each base model is trained on a portion of the dataset, either through random sampling or specific strategies such as k-fold cross-validation. The predictions of these base models are then stored for further use.
2. **Meta-Model Training:** In this stage, a meta-model, also called a blender or stacking model, is trained using the predictions from the base models. The meta-model learns to combine the base models' predictions and generate a final prediction. This can be achieved through multiple approaches, such as averaging the predictions, using a weighted sum, or training another model on top of the base model predictions.

The stacking ensemble method can provide better predictive performance compared to using individual models because it leverages the diversity of the base models. Each base model may have different strengths, weaknesses, and biases, and by combining their predictions, the stacking model can compensate for these differences and make more accurate predictions.

One important consideration in stacking is avoiding overfitting. Since the base models are trained on the same dataset, there is a risk of overfitting if the stacking model simply memorizes the base models' predictions. To mitigate this, techniques like cross-validation and regularization can be applied.

Overall, the stacking ensemble method is an effective methodology for combining multiple models to generate a more robust and accurate prediction model.

3.3.6 Performance Metrics

The performance of prediction model is evaluated based on certain evaluation

criteria and are generated through confusion matrix, which includes Accuracy, Recall, Precision, and Area Under Receiver Operating Characteristics Curve (AUC-ROC), F-measure etc. However, in this study Accuracy, one of the most widely used performance metric is used. [61] These metrics help to quantify the performance of machine learning models. [62]

3.3.6.1 Confusion Matrix

A specific table known as the confusion matrix is used to evaluate the effectiveness of machine learning algorithms. Each column of the matrix depicts the instance belonging to the predicted class whereas each row shows the actual class instance or vice versa. This matrix briefly represents the result given by the testing algorithm by providing a report of the number of True Positive (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). [63]

Prediction outcome

actual value		p	n	total
	p′	True Positive	False Negative	P′
	n′	False Positive	True Negative	N′
	total	P	N	

Figure 3.3: Confusion Matrix

3.3.6.2 Accuracy

The ratio of the correctly predicted instances to the total number of instances by the classifier is called accuracy. It measures the hit and miss of the classifier.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

3.3.6.3 Precision

Precision is defined as the percentage of accurately predicted positive instances by the classifier out of all positively classified instances.

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

3.3.6.4 Recall

The proportion of correctly classified positive examples to total positive instances for a given class. It is also referred to as true positive rate or sensitivity and it counts the number of hits of the classifier for the class.

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

3.3.6.5 F1 Score Measure

It is the harmonic mean of recall and precision.

$$Fmeasure = \frac{(2 * Recall * Precision)}{(Recall + Precision)} \tag{4}$$

3.4 Summary

This section concludes and briefly highlights the major parts of the proposed design. It includes the theoretical approach used to make this model. The design of the proposed model is discussed that involves preprocessing, feature engineering, transfer learning, handling class imbalance and ensemble learning method. The data sets, all of their features and then the feature engineering step is described. Moreover, the classifiers used in this model and the performance metrics which will be used to calculate the efficiency of model are also discussed. In short, the technical approach to make the software defect prediction model is described.

Implementation

4.1 Introduction

In this chapter the implementation details of proposed design are discussed. It gives details overview of used language, hardware, software requirements and other information related to model implementation.

4.2 Environment (Hardware and Software)

Google Colab environment is used for the implementation of the proposed model. Python 3.10.12 is used for implementation. 8 GB RAM with windows 10 on Core i7 PC is used in this research work. The libraries used in this model are sklearn, imblearn, pandas and numpy. For plotting the graphs matplotlib library is used.

4.3 Implementation Details

The implementation details of all steps mentioned in the previous chapter are discussed in detail as below:

4.3.1 Data Pre-Processing (Step 1)

To normalize the dataset, first the null values and Noise is checked and removed from the dataset.

4.3.1.1 Null values and Noise

The repository has null values and noise in the form of missing and duplicated rows. The null values are first identified and then removed by replacing with zero. Noise is reduced from the dataset by reducing redundant data, allowing our trained

algorithm to identify defects more correctly.

Implementation Steps:

To address the issue of Null, NAN, and noise in the dataset, the following steps are performed:

1. Read CSV File
2. Remove all unnecessary columns, i.e., name and version
3. Use panda’s isnull (). sum () method on the data retrieved from CSV file
4. This method traverses all columns and sum number of null values in them
5. Use panda’s fillna () method to fill the null values with 0.
6. Use panda’s isna () method on the data retrieved from CSV file
7. This method traverses all rows in the dataset and highlight the NAN values.
8. Use panda’s fillna () method to fill the NAN values with 0
9. Use panda’s duplicated method on the data retrieved from CSV file
10. Duplicating method of panda’s library, traversing each row in the dataset one by one across the file and picking the duplicated rows.
11. Use pandas’ method drop_duplicates on the data retrieved from the CSV file
12. Use drop_duplicates method to eliminate all the retrieved duplicated rows from the dataset.

4.3.1.2 Dataset Scaling

The data distribution gap in the datasets is filled after removing the null values and noise. Standard Scaler is a useful approach that is used as a preprocessing step to

standardize the range of functionality of the input dataset. Some factors assessed at different scales do not contribute equally to model fitting and model learned function and may result in bias. To address this potential issue, feature-wise standardized (μ=0, σ =1) is typically utilized prior to model fitting.

Implementation Steps:

1. Read source CSV File
2. Split the dataset into independed variables features and depended target variable.
3. Grouped the independed variables features set
4. Used the sklearn standardscaler feature and passed the independed variables features set to it
5. Standard Scaler removes the mean and scales each feature/variable to unit variance. This operation is performed feature-wise in an independent way.
6. The new scaled values of all independed features are saved in new data frame.
7. The implementation code of the above procedure is as follows:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_res)
```

4.3.1.3 Handling Multiclass Nature of Promise Dataset:

After removing null values and noise, the Promise dataset is converted to 2 class problem as this dataset is of multiclass nature. The multi-class nature of the dataset is described visually in the Figure 4.1 below.

The detailed analysis about classes in Promise Dataset reveals that most of the classes have much less data. To get better results, we kept class 0 but combined the classes with the bug labels 2,3,4,5,6,7 as 1. This indicated that Class 1 contained all of the bugs found in Classes 2, 3, 4, 5, 6, and 7. This step is performed to collect as much data as possible for training the model.

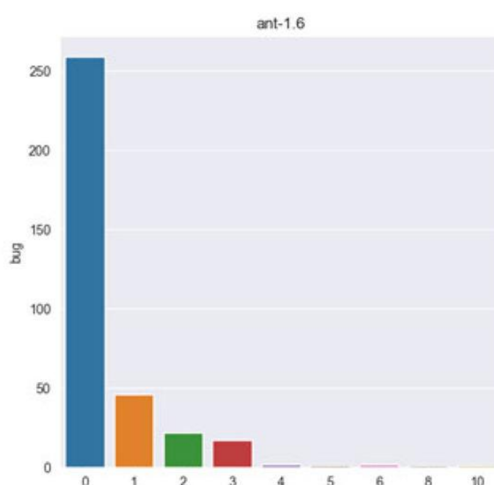


Figure 4.1: Multiclass Data of Ant Dataset

Implementation Steps:

To resolve the multiclass issue, following steps were performed:

1. Read CSV File of Promise dataset
2. Use NumPy's, where () method to traverse the target variable and highlight the values equal or greater than 1.
3. Replace all the highlighted values with 1in the CSV File.

4.3.2 Feature Engineering (Step 2)

For the efficient prediction of bugs by the proposed model, the concept of feature engineering has been incorporated.

Implementation Steps:

1. Read CSV File of dataset
2. Define a new variable named "eval"

3. Define a condition combining 5 existing features of a dataset, set a threshold value of each feature and used logical operator AND.
4. Insert a record in new feature when the condition of AND operator becomes "TRUE"
5. Convert the new feature to data frame and add it in the CSV file.
6. Transverse the newly added feature values and replace the True values with 1 and False values with 0.
7. The implementation code of the above procedure is as follows:

```
eval = (data.BRANCH_COUNT < 200) & (data.CONDITION_COUNT < 250) &
(data.CYCLOMATIC_COMPLEXITY < 100) & (data.DECISION_COUNT < 150)
& (data.NUMBER_OF_LINES < 500)
data['eval'] = pd.DataFrame(eval)
data['eval'] = [1 if e == True else 0 for e in data['eval']]
```

4.3.3 Class Imbalance (Step 3)

The proposed model also addresses the issue of class imbalance. In the proposed work, the issue of class imbalance is resolved both in terms of the overall number of instances and the total number of output classes. Random Over Sampler is used to tackle the class imbalance problem.

Implementation Steps:

The following steps are performed to resolve the class imbalance issue in the dataset:

1. Once the null and noise had been removed from the dataset, RandomOverSampler is used. It randomly selecting examples from the minority class, with replacement, and adding them to the training dataset.

2. The random state value is set to 26 and passed the dataset as independed variables features and target feature variable.

3. The dataset is passed and then, using this technique, all the minority classes were oversampled to the majority class.

4. The implementation code is as follows:

```
from imblearn.over_sampling import RandomOverSampler
os = RandomOverSampler(random_state = 26)
X_res, y_res = os.fit_resample(X,y)
```

The dataset results after applying class imbalance on both data repositories are shown below in table 4.1.

Table 4.1: Class Imbalance Results on Both Data Repositories

Sr.No.	Dataset	Dataset Class Info before Class Imbalance		Dataset Class Info After Class Imbalance	
		Yes	No	Yes	No
1	**CM1**	42	285	285	285
2	**MW1**	27	226	226	226
3	**PC1**	61	644	644	644
4	**PC3**	134	943	943	943
5	**PC4**	177	1110	1110	1110
6	**Ant-1.7**	166	579	579	579
7	**Camel-1.6**	188	777	777	777
8	**Ivy-2.0**	40	312	312	312
9	**Xalan-2.4**	110	613	613	613

4.3.4 Dataset Division (Step 4)

After features scaling and class imbalance, the dataset is divided into training and testing dataset. The 75% portion of the dataset is used for training and 25% for testing. Model is trained and tested on one project and then prediction is conducted on another dataset. The python code for dataset division is as follows:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_scaled , y_res, random_state =
1, test_size = 0.25)
```

4.3.5 Classification (Step 5)

Data classification is performed to ensure the accuracy of the suggested design. To accomplish this, various machine learning classifiers are used [15][56]. Decision tree, random forest, bagging, boosting, and stacking are selected for the classification of buggy data in datasets because the dataset is multi-class and proposed model will be applied on the cross-project. The classifier predicts the output using the mapped instances as training data. The data is classified into Class 0 and Class 1 using this classifier. The performance of each project is examined as a source and target, i.e., first picked CM1 as the source project and MW1 as the target project and performed prediction on this dataset and calculate accuracy. Then this step is repeated by changing all other datasets of NASA as target. The same process is repeated both data repositories which includes NASA and PROMISE and for all 9 datasets i.e., one dataset is source at a time and all others are target. But the datasets of NASA and PROMISE are treated separately due to different nature of features in both repositories.

The implementation details of one dataset are described below:

1. After data preprocessing, scaler transform, feature engineering and class imbalance, the dataset i.e., CM1 is divided into training and test dataset. The dataset is divided on the ratio of 75% for training and 25% for testing. By using the sklearn library and train_ test split function, the dataset is split into training and test dataset.

   ```
   from sklearn.model_selection import train_test_split
   X_train, X_test, y_train, y_test = train_test_split(X_scaled , y_res,
   random_state = 1, test_size = 0.25)
   ```

2. First Decision tree classifier is applied on training set of CM1 dataset to train the model. Sklearn library tree function is used to implement decision tree on training dataset.

```
dtree=DecisionTreeClassifier()
dtree.fit(X_train, y_train)
```

3. Then test dataset is passed to the model to predict the values by using decision tree predict property.

```
dtree.predict(X_test)
```

4. Then the same trained and tested model is passed to the ensemble classifier of bagging, AdaBoost and stacking.

5. In bagging method, sklearn.ensemble property is used to call bagging classifier. The base classifier is decision tree along with other parameters.

```
from sklearn.ensemble import BaggingClassifier
bag_model = BaggingClassifier(
    estimator = DecisionTreeClassifier(),
    n_estimators = 100,
    max_samples = 0.8,
    oob_score = True,
    random_state = 26
)
```

6. After model definition, the train dataset is passed to the classifier.
   `bag_model.fit(X_train,y_train)`

7. Then the test dataset is passed to the model for prediction.

```
y_pred = bag_model.predict(X_test)
```

8. The classification report is generated to check the accuracy of the model within a project. Sklearn.metrics property is used to call accuracy score and classification report.

```
# Evaluate the model
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
print (classification_report(y_test, y_pred))
print ("Accuracy: ",accuracy_score(y_pred ,y_test))
```

9. The prediction result is shown in figure 4.2, gives an insight into model prediction accuracy.

```
              precision    recall  f1-score   support

           0       1.00      0.86      0.92        57
           1       0.88      1.00      0.93        57

    accuracy                           0.93       114
   macro avg       0.94      0.93      0.93       114
weighted avg       0.94      0.93      0.93       114

Accuracy:  0.9298245614035088
```

Figure 4.2: Model Prediction Results Within Project

10. As the proposed model is for cross project defect prediction, so target project is loaded and split into independed variables features and depended target variable.

```
X1 = test_data.drop('Defective',axis='columns')
y1 = test_data.Defective
```

11. The independed variable features are passed to previously trained model for prediction.

```
y1_pred = bag_model.predict(X1)
```

12. The classification report is generated to check the accuracy of the model within a project. Sklearn.metrics property is used to call accuracy score and classification report.

```
# Evaluate the model
print (classification_report(y1, y1_pred))
```

print ("ACC: “, accuracy_score(y1_pred, y1))

13. The prediction results as shown in figure 4.3, gives an insight into model prediction accuracy.

```
              precision    recall  f1-score   support

           0       0.94      0.80      0.86       644
           1       0.17      0.43      0.24        61

    accuracy                           0.77       705
   macro avg       0.55      0.61      0.55       705
weighted avg       0.87      0.77      0.81       705

ACC:  0.7659574468085106
```

Figure 4.3: Model Prediction Results Across Projects

14. Steps as mentioned above from serial 5 to 13 are repeated for two others below mentioned ensemble methods and a base classifier i.e., Random Forest Classifier also.
15. For AdaBoost, the sklearn library is used to call Adaboost classifier.

ada=AdaBoostClassifier (base_estimator=dTree_clf, n_estimators = 500,)

16. For stacking classifier, different base classifiers are used which includes LinearSVC, MLP classifier and final estimator as decision tree. Sklearn library is used to call and define the classifier in python.

```
estimators = [
    ('svr', make_pipeline(StandardScaler(),
                          LinearSVC(random_state=1))),
    ('mlp’, make_pipeline(StandardScaler(), MLPClassifier(alpha=1, max_iter=200)))
]
stack_model = StackingClassifier(
    estimators=estimators,
    # final_estimator = RandomForestClassifier(random_state=1),
    final_estimator = DecisionTreeClassifier(random_state=1),
```

```
    cv=10
)
```

The same above procedure is followed for all classifiers on all datasets of both repositories.

4.4 Summary

This section concludes and briefly highlights the implementation details of the proposed design. It includes the tools used to implement this model. The implementation details of proposed model are discussed that involves preprocessing, feature engineering, transfer learning, handling class imbalance and ensemble learning method. The classifiers used in this model and the performance metrics which will be used to calculate the efficiency of model are described. In short, the practical approach to proposed model is discussed to make the software defect prediction model.

Results and Discussion

5.1 Introduction

In this chapter the model results will be discussed in detail. Different aspects of model prediction accuracy will be discussed to highlight the efficiency of the proposed model.

5.2 Implementation Details

Using the Random over sampling technique, the majority and minority classes in each dataset are balanced using the experimental framework. As in previous studies [55,59], the balanced representation of the classes was based on 50% defective and 50% non-defective classes. The goal is to ensure that the resulting models were trained with each class label and to provide credibility to the proposed model in predicting the proper class labels (defective or non-defective). Random Oversampling technique is used as a sampling approach because of its performance.

5.3 Results

This section presents and discusses the results received after evaluating the various classifiers. It is critical to demonstrate the significance of sampling technique on bug prediction model development. Furthermore, the efficacy of the class imbalance and ensemble approaches over the base-line classifier is a focus of this research.

5.3.1 Base Classifiers Results for NASA Dataset

At first the results will be presented to reflect the effects of each base-line classifier. Table 5.1 presents the prediction performances of base classifiers Decision Tree (DT)

and Random Forest (RF) on the datasets of NASA repository. The values shown the accuracy of the proposed prediction model employing both classifiers. The DT classifier, as seen in Table 5.1, yielded an average accuracy of 82% and RF have 84%. The RF classifier had highest prediction performances with an average accuracy of 84% as a base classifier.

Table 5.1: Prediction Performance of DT and RF on the NASA dataset

Sr.No	Dataset	Decision Tree (DT)	Random Forest (RF)
1	**PC1-PC3**	0.83	0.86
2	**PC1-PC4**	0.86	0.84
3	**PC1-CM1**	0.78	0.85
4	**PC1-MW1**	0.81	0.87
5	**PC3-PC1**	0.90	0.84
6	**PC3-PC4**	0.87	0.83
7	**PC3-CM1**	0.78	0.81
8	**PC3-MW1**	0.84	0.83
9	**PC4-PC1**	0.90	0.84
10	**PC4-PC3**	0.83	0.81
11	**PC4-CM1**	0.80	0.81
12	**PC4-MW1**	0.83	0.87
13	**CM1-PC1**	0.81	0.87
14	**CM1-PC3**	0.82	0.86
15	**CM1-PC4**	0.68	0.82
16	**CM1-MW1**	0.68	0.85
17	**MW1-PC1**	0.90	0.87
18	**MW1-PC3**	0.86	0.85
19	**MW1-PC4**	0.85	0.84
20	**MW1-CM1**	0.80	0.82
Average Accuracy %		**0.82**	**0.84**

5.3.2 Ensemble Methods along with Base Classifiers Results for NASA Dataset

Table 5.2 represents the proposed model prediction accuracy results on 2 individual classifiers and 3 different ensemble methods. Accuracy is a statistic that describes how the model performs in general across all classes. It is helpful when all classes are equally important. It is determined by dividing the number of right guesses by the total number of forecasts.

The method of bagging, boosting and stacking has been employed with base classifiers as DT and RF. The results shown that among the ensemble methods, boosting have the highest average accuracy with base classifier of RF. It has the average accuracy of 84% with lowest accuracy of 81% on PC3 and CM1 dataset and highest accuracy of 89% on PC1 dataset. CM1 dataset has less numbers of samples which may contribute to low accuracy of model prediction.

In the comparison of all three ensemble methods, stacking with DT have better average accuracy of 82%. Among bagging, BaggedDT has better results than BaggedRF. Where as in boosting, Ada boosted method is employed and AdaBoostRF yields better average prediction accuracy. The prediction results reveals that RF works better with proposed model, whether it's for individual classifier or for ensemble-based methods. The overall results indicate the model minimum accuracy up to 75%. The graph view of above results is shown in Figure 5.1.

Table 5.2: Prediction Performance of Ensemble Methods on NASA Dataset

Sr.No	Dataset	Decision Tree (DT)	Random Forest (RF)	Bagging		Boosting		Stacking	
				DT	RF	AdaBoost DT	AdaBoost RF	DT	RF
1	**PC1-PC3**	0.83	0.86	0.84	0.84	0.80	0.85	0.82	0.84
2	**PC1-PC4**	0.86	0.84	0.81	0.81	0.80	0.83	0.82	0.87
3	**PC1-CM1**	0.78	0.85	0.83	0.79	0.81	0.85	0.81	0.69
4	**PC1-MW1**	0.81	0.87	0.85	0.83	0.86	0.86	0.81	0.87
5	**PC3-PC1**	0.90	0.84	0.82	0.78	0.73	0.84	0.87	0.81
6	**PC3-PC4**	0.87	0.83	0.78	0.77	0.70	0.83	0.83	0.81

7	PC3-CM1	0.78	0.81	0.80	0.76	0.72	0.82	0.81	0.69
8	PC3-MW1	0.84	0.83	0.80	0.76	0.79	0.84	0.83	0.77
9	PC4-PC1	0.90	0.84	0.79	0.73	0.54	0.84	0.86	0.84
10	PC4-PC3	0.83	0.81	0.73	0.70	0.71	0.81	0.79	0.77
11	PC4-CM1	0.80	0.81	0.73	0.75	0.65	0.81	0.75	0.64
12	PC4-MW1	0.83	0.87	0.83	0.78	0.60	0.85	0.80	0.86
13	CM1-PC1	0.81	0.87	0.77	0.75	0.85	0.87	0.86	0.88
14	CM1-PC3	0.82	0.86	0.78	0.78	0.62	0.86	0.80	0.85
15	CM1-PC4	0.68	0.82	0.65	0.74	0.70	0.83	0.80	0.85
16	CM1-MW1	0.68	0.85	0.68	0.76	0.65	0.85	0.83	0.85
17	MW1-PC1	0.90	0.87	0.84	0.84	0.89	0.89	0.87	0.88
18	MW1-PC3	0.86	0.85	0.83	0.84	0.86	0.86	0.81	0.84
19	MW1-PC4	0.85	0.84	0.82	0.81	0.84	0.84	0.75	0.84
20	MW1-CM1	0.8	0.82	0.81	0.80	0.81	0.82	0.86	0.81
Average Accuracy %		**0.82**	**0.84**	**0.79**	**0.78**	**0.75**	**0.84**	**0.82**	**0.81**

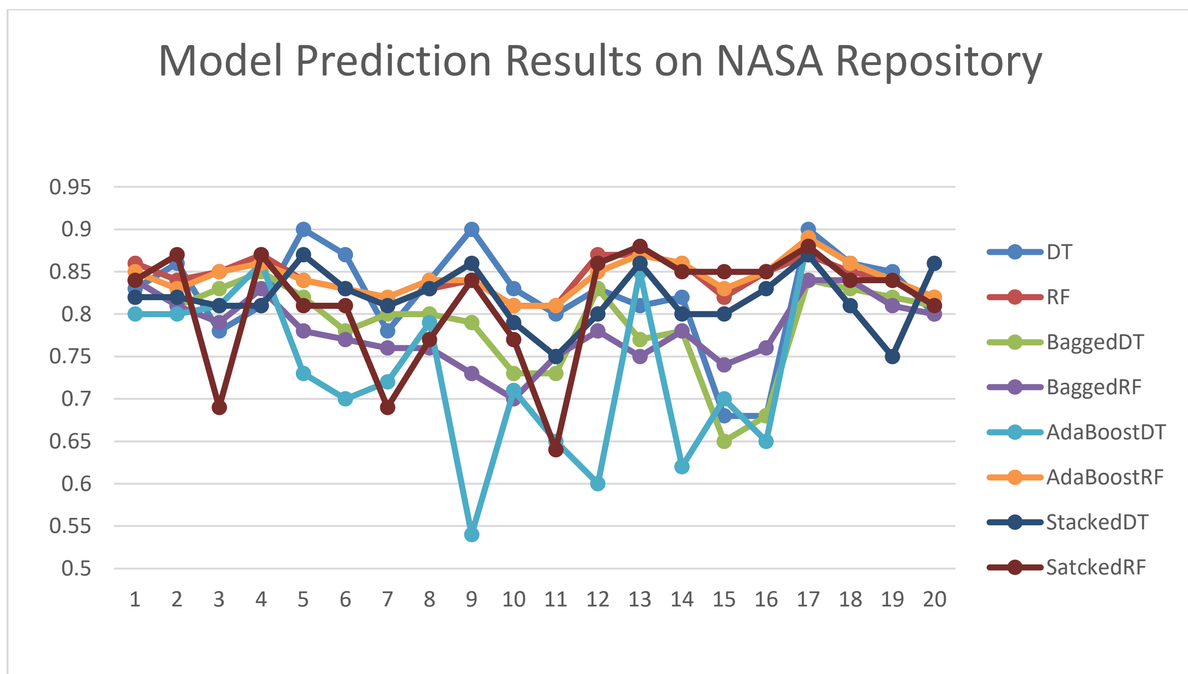


Figure 5.1: Model Prediction Results on NASA Repository

The results for the precision score of the proposed model are shown in Table 5.3.

The results highlights that an average precision of all classifiers is above 80% on all datasets. The precision is computed as the ratio of Positive samples that were correctly classified to all samples that were correctly or mistakenly identified as Positive. The precision measures how well the model categorizes a sample as positive.

Table 5.3: Precision Score of Proposed Model on NASA Dataset

Sr.No	Dataset	Decision Tree (DT)	Random Forest (RF)	Bagging		Boosting		Stacking	
				DT	RF	AdaBoost DT	AdaBoost RF	DT	RF
1	PC1-PC3	0.81	0.83	0.84	0.85	0.82	0.83	0.82	0.82
2	PC1-PC4	0.79	0.82	0.80	0.82	0.79	0.80	0.82	0.84
3	PC1-CM1	0.84	0.83	0.83	0.82	0.81	0.84	0.81	0.79
4	PC1-MW1	0.83	0.87	0.86	0.86	0.86	0.87	0.81	0.82
5	PC3-PC1	0.87	0.87	0.89	0.89	0.85	0.88	0.87	0.89
6	PC3-PC4	0.79	0.80	0.78	0.82	0.77	0.80	0.83	0.83
7	PC3-CM1	0.80	0.80	0.82	0.82	0.78	0.81	0.81	0.84
8	PC3-MW1	0.85	0.86	0.85	0.86	0.86	0.86	0.83	0.84
9	PC4-PC1	0.87	0.87	0.89	0.88	0.83	0.87	0.86	0.86
10	PC4-PC3	0.84	0.82	0.84	0.82	0.83	0.81	0.79	0.80
11	PC4-CM1	0.77	0.81	0.81	0.83	0.77	0.82	0.75	0.74
12	PC4-MW1	0.82	0.82	0.82	0.83	0.77	0.83	0.80	0.81
13	CM1-PC1	0.86	0.86	0.87	0.87	0.87	0.85	0.86	0.86
14	CM1-PC3	0.81	0.81	0.82	0.83	0.77	0.82	0.80	0.81
15	CM1-PC4	0.76	0.77	0.75	0.78	0.77	0.77	0.80	0.78
16	CM1-MW1	0.84	0.87	0.85	0.88	0.81	0.85	0.83	0.85
17	MW1-PC1	0.85	0.87	0.88	0.88	0.87	0.88	0.87	0.86
18	MW1-PC3	0.80	0.79	0.81	0.82	0.81	0.80	0.81	0.81
19	MW1-PC4	0.78	0.79	0.78	0.78	0.78	0.79	0.75	0.75
20	MW1-CM1	0.78	0.81	0.82	0.82	0.79	0.82	0.86	0.80
Average Precision %		**0.82**	**0.83**	**0.83**	**0.84**	**0.81**	**0.83**	**0.82**	**0.82**

In the next table 5.4, the recall score for the model prediction results has been shown.

The recall of the model assesses its ability to detect Positive samples. The more positive samples identified, the larger the recall.

Table 5.4: Recall Score of Proposed Model on NASA Dataset

Sr.No	Dataset	Decision Tree (DT)	Random Forest (RF)	Bagging		Boosting		Stacking	
				DT	RF	AdaBoost DT	AdaBoost RF	DT	RF
1	PC1-PC3	0.80	0.85	0.84	0.84	0.80	0.85	0.81	0.81
2	PC1-PC4	0.81	0.84	0.81	0.81	0.80	0.83	0.84	0.86
3	PC1-CM1	0.84	0.84	0.83	0.79	0.81	0.85	0.67	0.63
4	PC1-MW1	0.82	0.86	0.85	0.83	0.86	0.86	0.80	0.83
5	PC3-PC1	0.78	0.82	0.82	0.77	0.73	0.84	0.77	0.76
6	PC3-PC4	0.76	0.83	0.78	0.77	0.70	0.83	0.81	0.81
7	PC3-CM1	0.78	0.82	0.80	0.76	0.72	0.82	0.58	0.56
8	PC3-MW1	0.83	0.84	0.80	0.76	0.79	0.84	0.67	0.68
9	PC4-PC1	0.83	0.84	0.79	0.73	0.54	0.84	0.82	0.82
10	PC4-PC3	0.78	0.78	0.84	0.70	0.71	0.81	0.75	0.75
11	PC4-CM1	0.76	0.79	0.73	0.75	0.65	0.81	0.63	0.62
12	PC4-MW1	0.85	0.85	0.83	0.78	0.60	0.85	0.79	0.77
13	CM1-PC1	0.77	0.85	0.77	0.75	0.85	0.87	0.84	0.87
14	CM1-PC3	0.74	0.84	0.73	0.78	0.62	0.86	0.81	0.85
15	CM1-PC4	0.67	0.81	0.65	0.74	0.70	0.83	0.85	0.85
16	CM1-MW1	0.68	0.85	0.66	0.76	0.65	0.85	0.84	0.88
17	MW1-PC1	0.89	0.87	0.84	0.84	0.89	0.89	0.88	0.89
18	MW1-PC3	0.86	0.84	0.83	0.84	0.86	0.86	0.84	0.86
19	MW1-PC4	0.85	0.84	0.82	0.81	0.84	0.84	0.83	0.85
20	MW1-CM1	0.82	0.82	0.81	0.80	0.81	0.82	0.84	0.83
Average Recall %		**0.80**	**0.83**	**0.79**	**0.78**	**0.75**	**0.84**	**0.78**	**0.79**

Ideally, both precision and recall metrics should be maximized to obtain the perfect classifier. The average recall of model ranges from 0.75 to 0.84.

The F1 score combines precision and recall by using their harmonic mean, thus

maximizing the F1 score implies maximizing both precision and recall at the same time. As a result, researchers have chosen the F1 score to evaluate their models in conjunction with accuracy. The F1 score of the proposed model has been shown in Table 5.5. It ranges from 0-100%, and a higher F1 score denotes a better-quality classifier. The results indicate that F1 score of a proposed model is greater than 75% for all individual and ensemble classifiers.

Table 5.5: F1-Score of Proposed Model on NASA Dataset

Sr.No	Dataset	Decision Tree (DT)	Random Forest (RF)	Bagging		Boosting		Stacking	
				DT	RF	AdaBoost DT	AdaBoost RF	DT	RF
1	PC1-PC3	0.81	0.84	0.84	0.84	0.81	0.84	0.81	0.82
2	PC1-PC4	0.80	0.82	0.81	0.81	0.80	0.81	0.83	0.85
3	PC1-CM1	0.84	0.83	0.83	0.80	0.81	0.84	0.72	0.69
4	PC1-MW1	0.83	0.86	0.86	0.85	0.86	0.86	0.81	0.83
5	PC3-PC1	0.82	0.84	0.85	0.81	0.78	0.86	0.81	0.81
6	PC3-PC4	0.77	0.81	0.78	0.79	0.73	0.81	0.82	0.82
7	PC3-CM1	0.79	0.81	0.81	0.78	0.75	0.81	0.65	0.63
8	PC3-MW1	0.84	0.85	0.82	0.80	0.82	0.85	0.73	0.74
9	PC4-PC1	0.85	0.85	0.83	0.79	0.64	0.85	0.84	0.84
10	PC4-PC3	0.80	0.80	0.77	0.74	0.76	0.81	0.77	0.77
11	PC4-CM1	0.77	0.80	0.77	0.78	0.70	0.82	0.68	0.67
12	PC4-MW1	0.83	0.83	0.82	0.80	0.67	0.84	0.80	0.79
13	CM1-PC1	0.81	0.85	0.81	0.79	0.85	0.86	0.85	0.87
14	CM1-PC3	0.77	0.83	0.77	0.80	0.68	0.83	0.81	0.83
15	CM1-PC4	0.71	0.79	0.69	0.76	0.73	0.79	0.81	0.80
16	CM1-MW1	0.74	0.86	0.72	0.80	0.71	0.85	0.83	0.86
17	MW1-PC1	0.87	0.87	0.86	0.86	0.88	0.89	0.88	0.87
18	MW1-PC3	0.82	0.81	0.82	0.83	0.83	0.82	0.82	0.83
19	MW1-PC4	0.80	0.81	0.80	0.79	0.80	0.81	0.79	0.79
20	MW1-CM1	0.80	0.81	0.81	0.81	0.80	0.82	0.85	0.81
Average F1-Score %		**0.80**	**0.83**	**0.80**	**0.80**	**0.77**	**0.83**	**0.80**	**0.80**

5.3.3 Base Classifiers Results for PROMISE Dataset

In the next step, the same proposed model is used for prediction on PROMISE dataset to check the model accuracy. Table 5.6 represents the prediction results of proposed model on PROMISE dataset for individual classifiers of DT and RF. The prediction result reveals that RF works better than DT. RF have an average accuracy of 77% with highest accuracy of 83% on ivy-2.0 dataset and lowest accuracy of 66% on Camel -1.6 dataset.

Table 5.6: Prediction Performance of DT and RF on PROMISE dataset

Sr.No	Dataset	Decision Tree	Random Forest
1	Ant-1.7-Camel-1.6	0.55	0.66
2	Ant-1.7-Ivy-2.0	0.72	0.80
3	Ant-1.7-Xalan-2.4	0.69	0.80
4	Camel-1.6- Ant-1.7-	0.73	0.74
5	Camel-1.6- Ivy-2.0	0.76	0.83
6	Camel-1.6- Xalan-2.4	0.73	0.79
7	Ivy-2.0- Ant-1.7	0.71	0.81
8	Ivy-2.0- Camel-1.6	0.70	0.75
9	Ivy-2.0- Xalan-2.4	0.72	0.82
10	Xalan-2.4- Ant-1.7	0.63	0.73
11	Xalan-2.4- Camel-1.6	0.64	0.69
12	Xalan-2.4- Ivy-2.0	0.72	0.76
Average Accuracy %		**0.69**	**0.77**

5.3.4 Ensemble Methods along with Base Classifiers Results for PROMISE Dataset

After individual classifiers prediction, the model is tested using 3 ensemble methods for PROMISE Dataset. Table 5.7, shows the prediction results of a proposed model for PROMISE datsets. The prediction results reveal that, model performance is around 65% for all ensemble methods with both base classifiers. In Bagging, BaggedDT have high average accuracy of 71% with highest accuracy of 79% on

Xalan-2.4 dataset and lowest of 60% on Camel-1.6 dataset. BaggedRF have average accuracy of 70% with highest on Ivy-2.0 dataset and lowest on Camel-1.6 dataset. In Boosting, AdaBoostRF have high accuracy of 76% with highest accuracy of 83% on Ivy-2.0 and Xalan-2.4 dataset. AdaboostDT have average accuracy of 64% with highest on Xalan-2.4 dataset and lowest on Camel-1.6. Whereas for Stacking, again SatckedRF have highest average accuracy of 79%. Both base classifiers have highest accuracy for Ivy-2.0 Dataset.

Table 5.7: Prediction Performance of Ensemble Methods on PROMISE datasets

Sr.No	Dataset	Decision Tree (DT)	Random Forest (RF)	Bagging		Boosting		Stacking	
				DT	RF	AdaBoost DT	AdaBoost RF	DT	RF
1	**Ant-1.7-Camel-1.6**	0.55	0.66	0.60	0.61	0.60	0.64	0.72	0.74
2	**Ant-1.7-Ivy-2.0**	0.72	0.80	0.72	0.75	0.59	0.80	0.81	0.86
3	**Ant-1.7-Xalan-2.4**	0.69	0.80	0.71	0.75	0.58	0.77	0.74	0.82
4	**Camel-1.6- Ant-1.7-**	0.73	0.74	0.70	0.68	0.70	0.73	0.67	0.75
5	**Camel-1.6- Ivy-2.0**	0.76	0.83	0.74	0.79	0.69	0.83	0.75	0.79
6	**Camel-1.6- Xalan-2.4**	0.73	0.79	0.79	0.78	0.72	0.80	0.72	0.81
7	**Ivy-2.0- Ant-1.7**	0.71	0.81	0.76	0.73	0.62	0.79	0.76	0.82
8	**Ivy-2.0- Camel-1.6**	0.70	0.75	0.63	0.65	0.66	0.75	0.75	0.73
9	**Ivy-2.0- Xalan-2.4**	0.72	0.82	0.72	0.77	0.73	0.83	0.78	0.80
10	**Xalan-2.4- Ant-1.7**	0.63	0.73	0.70	0.66	0.65	0.72	0.73	0.76
11	**Xalan-2.4- Camel-1.6**	0.64	0.69	0.65	0.60	0.53	0.66	0.69	0.74
12	**Xalan-2.4- Ivy-2.0**	0.72	0.76	0.74	0.67	0.62	0.75	0.79	0.84
Average Accuracy %		**0.60**	**0.77**	**0.71**	**0.70**	**0.64**	**0.76**	**0.74**	**0.79**

The model prediction results on PROMISE repository indicates the stable nature of model prediction as there is no huge variation in the prediction results on different datasets. The graph view of Table 5.7 is shown in Figure 5.2.

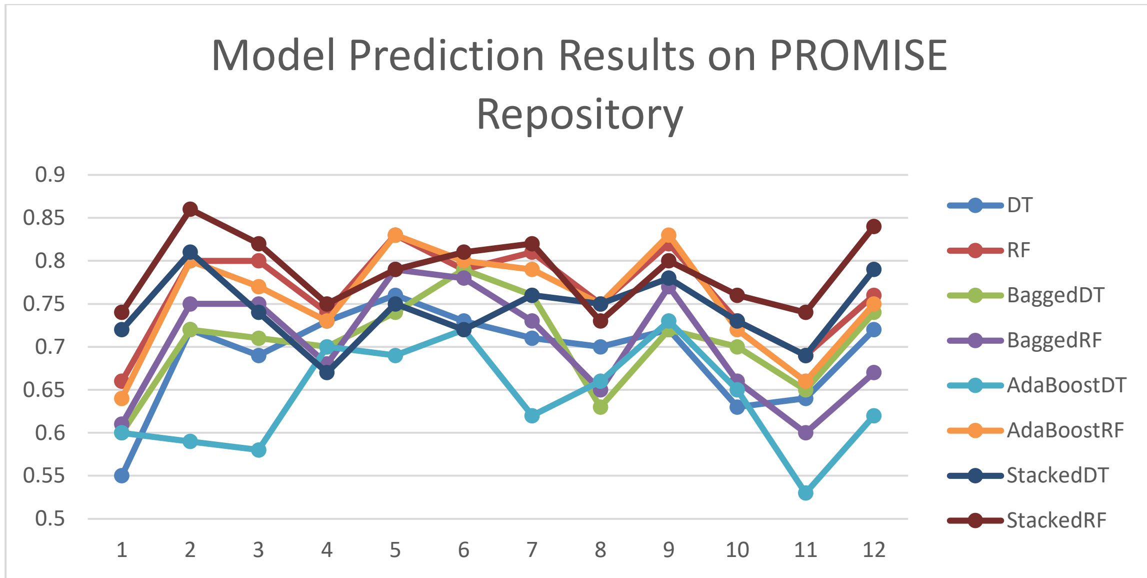


Figure 5.2: Model Prediction Results on PROMISE Repository

The model prediction results for precision, recall and F1 score on the RPOMISE repository has been shown in the Table 5.8, 5.9 and 5.10 respectively.

Table 5.8: Precision Score of Proposed Model on PROMISE Dataset

Sr.No	Dataset	Decision Tree (DT)	Random Forest (RF)	Bagging		Boosting		Stacking	
				DT	RF	AdaBoost DT	AdaBoost RF	DT	RF
1	**Ant-1.7-Camel-1.6**	0.68	0.72	0.71	0.72	0.72	0.72	0.71	0.72
2	**Ant-1.7-Ivy-2.0**	0.81	0.89	0.87	0.88	0.84	0.89	0.86	0.88
3	**Ant-1.7-Xalan-2.4**	0.84	0.82	0.87	0.84	0.75	0.83	0.79	0.82
4	**Camel-1.6- Ant-1.7-**	0.68	0.72	0.72	0.74	0.69	0.72	0.67	0.71
5	**Camel-1.6- Ivy-2.0**	0.81	0.85	0.81	0.85	0.80	0.85	0.81	0.83
6	**Camel-1.6- Xalan-2.4**	0.77	0.78	0.78	0.80	0.77	0.79	0.76	0.79
7	**Ivy-2.0- Ant-1.7**	0.72	0.81	0.80	0.81	0.71	0.79	0.73	0.80
8	**Ivy-2.0- Camel-1.6**	0.70	0.74	0.71	0.73	0.71	0.73	0.72	0.71
9	**Ivy-2.0- Xalan-2.4**	0.78	0.81	0.80	0.83	0.79	0.81	0.76	0.81
10	**Xalan-2.4- Ant-1.7**	0.69	0.77	0.74	0.78	0.69	0.75	0.71	0.73
11	**Xalan-2.4- Camel-1.6**	0.67	0.72	0.70	0.73	0.68	0.71	0.70	0.71
12	**Xalan-2.4- Ivy-2.0**	0.81	0.84	0.82	0.87	0.82	0.86	0.83	0.82
Average Precision %		**0.75**	**0.79**	**0.78**	**0.80**	**0.75**	**0.79**	**0.75**	**0.78**

Table 5.9: Recall Score of Proposed Model on PROMISE Dataset

Sr.No	Dataset	Decision Tree (DT)	Random Forest (RF)	Bagging		Boosting		Stacking	
				DT	RF	AdaBoost DT	AdaBoost RF	DT	RF
1	**Ant-1.7-Camel-1.6**	0.55	0.66	0.60	0.61	0.60	0.64	0.72	0.74
2	**Ant-1.7-Ivy-2.0**	0.52	0.80	0.72	0.75	0.59	0.80	0.81	0.86
3	**Ant-1.7-Xalan-2.4**	0.72	0.79	0.72	0.75	0.58	0.77	0.74	0.82
4	**Camel-1.6- Ant-1.7-**	0.68	0.73	0.70	0.68	0.70	0.73	0.67	0.75
5	**Camel-1.6- Ivy-2.0**	0.75	0.83	0.74	0.79	0.69	0.83	0.75	0.79
6	**Camel-1.6- Xalan-2.4**	0.73	0.79	0.79	0.78	0.72	0.80	0.72	0.81
7	**Ivy-2.0- Ant-1.7**	0.71	0.81	0.76	0.73	0.62	0.79	0.76	0.82
8	**Ivy-2.0- Camel-1.6**	0.68	0.75	0.63	0.65	0.66	0.75	0.75	0.73
9	**Ivy-2.0- Xalan-2.4**	0.72	0.82	0.72	0.77	0.73	0.82	0.78	0.80
10	**Xalan-2.4- Ant-1.7**	0.63	0.73	0.70	0.66	0.65	0.72	0.73	0.76
11	**Xalan-2.4- Camel-1.6**	0.64	0.69	0.65	0.60	0.53	0.66	0.69	0.74
12	**Xalan-2.4- Ivy-2.0**	0.72	0.76	0.74	0.67	0.62	0.75	0.79	0.84
Average Recall %		**0.67**	**0.76**	**0.71**	**0.70**	**0.64**	**0.76**	**0.74**	**0.79**

Table 5.10: F1-Score of Proposed Model on PROMISE Dataset

Sr.No	Dataset	Decision Tree (DT)	Random Forest (RF)	Bagging		Boosting		Stacking	
				DT	RF	AdaBoost DT	AdaBoost RF	DT	RF
1	**Ant-1.7-Camel-1.6**	0.60	0.68	0.64	0.65	0.64	0.67	0.71	0.73
2	**Ant-1.7-Ivy-2.0**	0.61	0.82	0.77	0.79	0.66	0.83	0.83	0.87
3	**Ant-1.7-Xalan-2.4**	0.77	0.80	0.77	0.78	0.64	0.79	0.76	0.82
4	**Camel-1.6- Ant-1.7-**	0.68	0.73	0.71	0.70	0.69	0.72	0.67	0.72
5	**Camel-1.6- Ivy-2.0**	0.78	0.84	0.77	0.81	0.74	0.84	0.77	0.81
6	**Camel-1.6- Xalan-2.4**	0.75	0.78	0.79	0.79	0.74	0.80	0.74	0.80

7	Ivy-2.0- Ant-1.7	0.71	0.81	0.77	0.75	0.65	0.79	0.74	0.81
8	Ivy-2.0- Camel-1.6	0.69	0.74	0.66	0.68	0.68	0.74	0.73	0.72
9	Ivy-2.0- Xalan-2.4	0.75	0.82	0.75	0.79	0.76	0.82	0.77	0.80
10	Xalan-2.4- Ant-1.7	0.65	0.74	0.72	0.69	0.66	0.73	0.72	0.74
11	Xalan-2.4- Camel-1.6	0.65	0.71	0.67	0.63	0.58	0.68	0.70	0.72
12	Xalan-2.4- Ivy-2.0	0.76	0.79	0.77	0.73	0.69	0.79	0.81	0.83
Average F1-Score %		**0.70**	**0.77**	**0.73**	**0.73**	**0.68**	**0.77**	**0.75**	**0.78**

5.3.5 Recall Comparison with and without Class Imbalance

Another experimental results of model prediction shown that model average accuracy without employing class imbalance is high i.e., 87% for random forest (RF) for cross project defect prediction as compared to with class imbalance. But the careful analysis of other performance metrics like precision, recall and F1 score indicates that average recall of buggy class without handling class imbalance is 0.05 which indicates that model prediction for buggy class is not efficient and model is overfitting the results by neglecting the buggy class instances. The class imbalance resolves this issue by generating equal samples of minority class for efficient model prediction. After employing the class imbalance, the recall of buggy class becomes 0.27. The analysis of the same is shown in Table 5.11.

Table 5.11: Prediction Analysis with and Without Class Imbalance

Sr. No	Dataset	Without Class Imbalance				With Class Imbalance			
		RF Accuracy	Precision of Buggy Class	Recall of Buggy Class	F1 Score of Buggy Class	RF Accuracy	Precision of Buggy Class	Recall of Buggy Class	F1 Score of Buggy Class
1	PC1-PC3	0.87	0.21	0.02	0.04	0.86	0.39	0.25	0.31
2	PC1-PC4	0.86	0.57	0.05	0.08	0.84	0.36	0.23	0.28
3	PC1-CM1	0.87	0.33	0.05	0.08	0.85	0.32	0.29	0.30
4	PC1-MW1	0.88	0.17	0.04	0.06	0.87	0.39	0.44	0.41

5	**PC3-PC1**	0.91	0.43	0.15	0.22	0.84	0.21	0.39	0.28
6	**PC3-PC4**	0.86	0.44	0.05	0.08	0.83	0.28	0.19	0.23
7	**PC3-CM1**	0.86	0.25	0.05	0.08	0.81	0.26	0.26	0.26
8	**PC3-MW1**	0.88	0.17	0.04	0.06	0.83	0.29	0.44	0.35
9	**PC4-PC1**	0.91	0.40	0.16	0.23	0.84	0.2	0.31	0.24
10	**PC4-PC3**	0.86	0.28	0.07	0.11	0.81	0.22	0.29	0.25
11	**PC4-CM1**	0.85	0	0	0	0.81	0.26	0.26	0.26
12	**PC4-MW1**	0.89	0	0	0	0.87	0.35	0.22	0.27
13	**CM1-PC1**	0.91	0.3	0.05	0.08	0.87	0.24	0.21	0.22
14	**CM1-PC3**	0.87	0.44	0.03	0.06	0.86	0.31	0.13	0.18
15	**CM1-PC4**	0.85	0.13	0.02	0.04	0.82	0.21	0.11	0.14
16	**CM1-MW1**	0.85	0.13	0.02	0.04	0.85	0.34	0.41	0.37
17	**MW1-PC1**	0.91	0.55	0.1	0.17	0.87	0.24	0.39	0.30
18	**MW1-PC3**	0.87	0.33	0.02	0.04	0.85	0.25	0.1	0.15
19	**MW1-PC4**	0.85	0.17	0.02	0.03	0.84	0.29	0.11	0.16
20	**MW1-CM1**	0.87	0.40	0.05	0.09	0.82	0.29	0.29	0.29
Average		**0.87**	**0.29**	**0.05**	**0.08**	**0.84**	**0.29**	**0.27**	**0.26**

5.3.6 Average Comparison Results on Both Repositories

The average comparison of all classifiers employed for model prediction is showed in table 5.12 and 5.13 for NASA and PROMISE repository respectively. The prediction results shown that RF have highest average accuracy on both data repositories. The graph view of the same is shown in figure 5.3 and 5.4.

Table 5.12: Average Prediction Performance Comparison for NASA Repository

Prediction Models	**Average Accuracy %**
Decision Tree (DT)	82
Random Forest (RF)	84
BaggedDT	79

BaggedRF	78
AdaBoostDT	75
AdaBoostRF	84
StackedDT	82
StackedRF	81

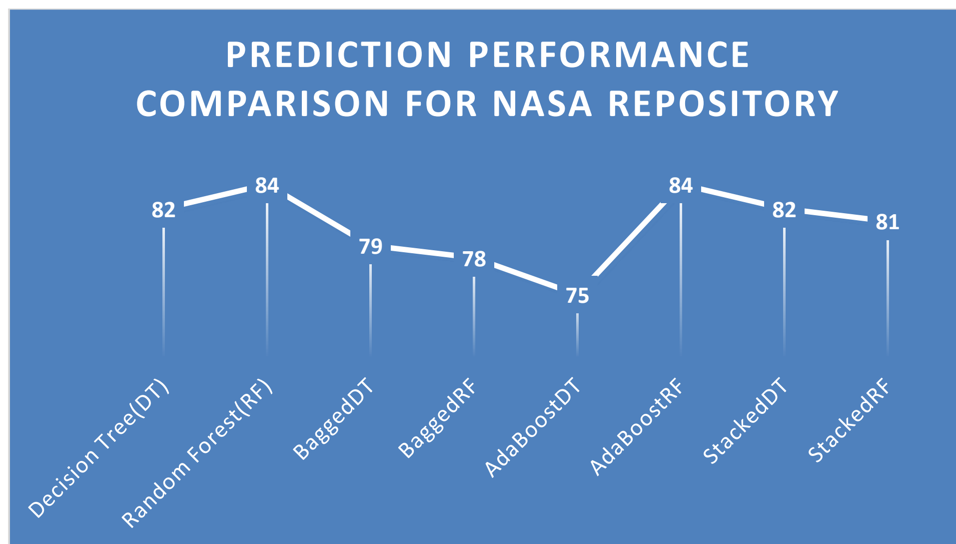


Figure 5.3: Average Prediction Performance Comparison for NASA Repository

Table 5.13: Average Prediction Performance Comparison for PROMISE Repository

Prediction Models	**Average Accuracy %**
Decision Tree (DT)	69
Random Forest (RF)	77
BaggedDT	71
BaggedRF	70
AdaBoostDT	64
AdaBoostRF	76
StackedDT	74
StackedRF	79

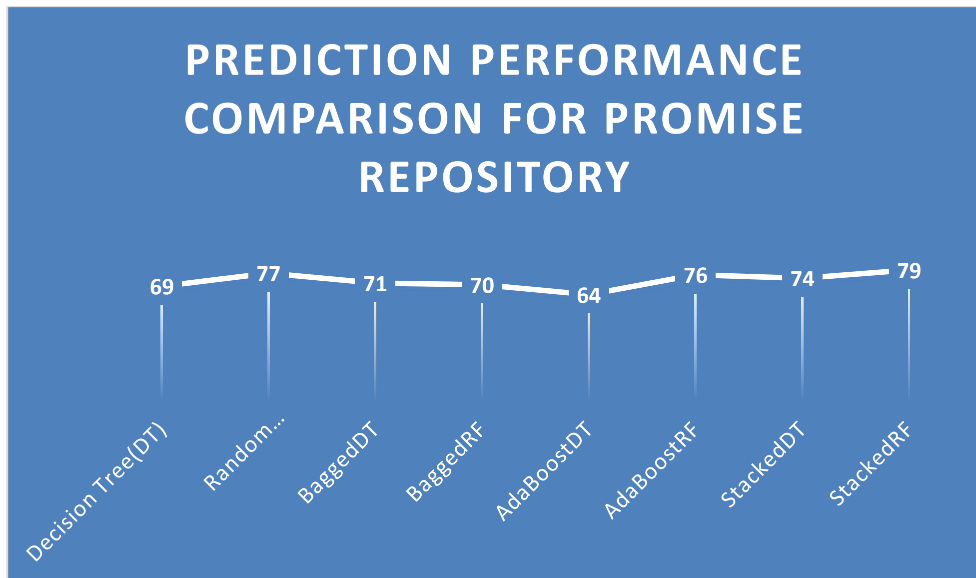


Figure 5.4: Average Prediction Performance Comparison for PROMISE Repository

5.4 Summary

The result of different experiments on NASA and Promise repository has been conducted to evaluate the model accuracy along with other software metrics like precision, recall and F1 score. The experimental result reveals that RF works best on the proposed model both as individual classifier and also for ensemble methods of bagging, boosting and stacking. The recall of buggy class also calculated to evaluate the model performance.

Conclusion

6.1 Introduction

In this chapter the conclusion of the proposed study will be discussed in detail. The results and future work will be discussed.

6.2 Conclusion

Software defect prediction is critical for detecting flaws early in the software development life cycle. This early detection and eradication of software flaws is critical for producing a cost-effective and high-quality software product. Though prior research has effectively used machine learning approaches for software defect prediction, when applied to imbalanced data sets, these techniques provide biased results. An imbalanced data collection has a non-uniform class distribution, with very few instances of one class compared to the other. The use of skewed datasets results in off-target predictions of the minority class, which is often thought to be more important than the majority class. Thus, efficiently handling unbalanced data is critical for the successful creation of a competent defect prediction model.

This work is based on design of an efficient model for software bug prediction in cross project software's by handling class imbalance issue. The random oversampling strategy was used to mitigate the data imbalance issues. The proposed designed model is implemented in Python. 2 individual and 3 ensemble classifiers are used to evaluate the model prediction accuracy across projects. The datasets of two data repositories i.e., NASA and PROMISE are used. The performance of different classifiers is evaluated using classification accuracy, Precision, Recall and F1-measure metrics. The outcomes of the conducted experiment showed that Random Forest (RF) performed well both as in individual classifier and also in ensemble methods. RF have an average accuracy of 84% for

NASA and 77% for PROMISE repository. For ensemble methods AdaBoostRF has highest accuracy of 84% on NASA datasets and 79% with StackedRF on PROMISE datasets. One other statics highlights that recall of a buggy class with imbalance methods is high as compared with non-imbalance handling.

6.3 Future Work

The proposed work can be used to study the impact of different sampling strategies like SMOTE, and under sampling. Similarly, an intriguing future addition could include investigating the influence of various feature selection methodologies in order to select the best set of features for software defect prediction. One future direction is to investigate and compare the performance of ensemble classifiers with alternative resampling strategies, as data imbalance continues to be a problem that degrades the effectiveness of existing software defect prediction systems. Similarly, the proposed model can also be employed for other datasets to study the model prediction accuracy on them.

6.4 Summary

The conclusion of the proposed model has been discussed in detail The results reveals that model works best with RF classifier for both datasets. RF classification in ensemble methods is also outperforming DT. In the last the future direction of the proposed model has been proposed.

References

[1] Syahana Nur'Ain Saharudin, Koh Tieng Wei, and Kew Si Na. Machine learning techniques for software bug prediction: A systematic review. Journal of Computer

[2] Siers MJ, Md ZI (2015) Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem. Inf Syst 51:62–71

[3] Laradji IH, Alshayeb M, Ghouti L (2015) Software defect prediction using ensemble learning on selected features. Inf Softw Technol 58:388–402

[4] Tong H, Liu B, Wang S (2018) Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. Inf Softw Technol 96:94–111

[5] Yang X, Lo D, Xia X, Sun J (2017) Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. Inf Softw Technol 87:206–220

[6] Pandey SK, Mishra RB, Tripathi AK (2020) Bpdet: An effective software bug prediction model using deep representation and ensemble learning techniques. Expert Syst Appl 144:113085

[7] Moustafa S, ElNainay MY, El Makky N, Abougabal MS (2018) Software bug prediction using weighted majority voting techniques. Alexandria Eng J 57(4):2763–2774

[8] Shanthini A (2014) Effect of ensemble methods for software fault prediction at various metrics level

[9] Hussain S, Keung J, Khan AA, Bennin KE (2015) Performance evaluation of ensemble methods for software fault prediction: An experiment. In: Proceedings of the ASWEC 2015 24th Australasian software engineering conference, pp 91–95

[10] Petrić J, Bowes D, Hall T, Christianson B, Baddoo N (2016) Building an ensemble for software defect prediction based on diversity selection. In: Proceedings of the 10th ACM/IEEE International symposium on empirical software engineering and measurement, pp 1–10

[11] Li R, Zhou L, Zhang S, Liu H, Huang X, Sun Z (2019) Software defect prediction based on ensemble learning. In: Proceedings of the 2019 2nd International conference on data science and information technology, pp 1–6

[12] Yohannese CW, Li T, Bashir K (2018) A three-stage based ensemble learning for improved software fault prediction: an empirical comparative study. Int J Comput Intell Sys 11(1):1229–1247

[13] Alsawalqah H, Hijazi N, Eshtay M, Faris H, Radaideh AA, Aljarah I, Alshamaileh Y (2020) Software defect prediction using heterogeneous ensemble classification based on segmented patterns. Appl Sci 10(5):1745

[14] Abdou AS, Darwish NR (2018) Early prediction of software defect using ensemble learning: A comparative study. Int J Comput Appl 179(46)

[15] Khuat TT, Le MH (2020) Evaluation of sampling-based ensembles of classifiers on imbalanced data for software defect prediction problems. SN Computer Science 1:1–16

[16] Twala B (2011) Predicting software faults in large space systems using machine learning techniques

[17] Ryu D, Jang Jong-In, Baik J (2017) A transfer cost-sensitive boosting approach for cross-project defect prediction. Softw Qual J 25(1):235–272

[18] Saifudin A, Hendric SWHL, Soewito B, Gaol FL, Abdurachman E, Heryadi Y (2019) Tackling imbalanced class on cross-project defect prediction using ensemble smote. In: IOP conference series: Materials science and engineering, vol 662. IOP Publishing

[19] Wang T, Zhang Z, Jing X, Zhang L (2016) Multiple kernel ensemble learning for software defect prediction. Autom Softw Eng 23(4):569–590

[20] Li N, Li Z, Nie Y, Sun X, Li X (2011) Predicting software black-box defects using stacked generalization. In: 2011 Sixth International conference on digital information management. IEEE, pp 294–299

[21] Sun Z, Song Q, Zhu X (2012) Using coding-based ensemble learning to improve software defect prediction. IEEE Trans Sys Man Cybern Part C (Applications and Reviews) 42(6):1806–1817

[22] Rathore SS, Kumar S (2016) Ensemble methods for the prediction of number of faults A study on eclipse project. In: 2016 11th International Conference on Industrial and Information Systems (ICIIS). IEEE, pp 540–545

[23] Yohannese CW, Li T, Simfukwe M, Khurshid F (2017) Ensembles based combined learning for improved software fault prediction: A comparative study. In 2017 12th International conference on intelligent systems and knowledge engineering (ISKE). IEEE, pp 1–6

[24] Bal PR, Kumar S (2018) Extreme learning machine based linear homogeneous ensemble for software fault prediction. In: ICSOFT, pp 103–112

[25] Mousavi R, Eftekhari M, Rahdari F (2018) Omni-ensemble learning (oel): Utilizing over-bagging, static and dynamic ensemble selection approaches for software defect prediction. Int J Artif Intell Tools 27 (06):1850024

[26] Campos JR, Costa E, Vieira M (2019) Improving failure prediction by ensembling the decisions of machine learning models: A case study. IEEE Access 7:177661–177674

[27] He H, Zhang X, Wang Q, Ren J, Liu J, Zhao X, Cheng Y (2019) Ensemble multiboost based on ripper classifier for prediction of imbalanced software defect data. IEEE Access 7:110333–110343

[28] Malhotra R, Jain J (2020) Handling imbalanced data using ensemble learning in software defect prediction. In: 2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence). IEEE, pp 300–304

[29] Zheng J (2010) Cost-sensitive boosting neural networks for software defect prediction. Expert Syst Appl 37(6):4537–4543

[30] Kumar L, Rath S, Sureka A (2017) Using source code metrics and ensemble methods for fault proneness prediction. arXiv:1704.04383

[31] Gao Y, Yang C (2016) Software defect prediction based on adaboost algorithm under imbalance distribution. In: 2016 4th International Conference on Sensors, Mechatronics and Automation (ICSMA 2016). Atlantis Press

[32] Coelho RA, dos RN Guimarães F, Esmin AAA (2014) Applying swarm ensemble clustering technique for fault prediction using software metrics. In: 2014 13th International conference on machine learning and applications. IEEE, pp 356–361

[33] Ryu D, Baik J (2018) Effective harmony search-based optimization of cost-sensitive boosting for improving the performance of cross-project defect prediction. KIPS Trans Softw Data Eng 7(3):77–90

[34] Jonsson L, Borg M, Broman D, Sandahl K, Eldh S, Runeson P (2016) Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. Empir Softw Eng 21(4):1533–1578

[35] Li Z, Jing X-Y, Zhu X, Zhang H, Xu B, Ying S (2019) Heterogeneous defect prediction with two-stage ensemble learning. Autom Softw Eng 26(3):599–651

[36] Mısırlı AT, Bener A, Turhan B (2011) An industrial case study of classifier ensembles for locating software defects. Softw Qual J 19(3):515–536

[37] Ryu D, Choi O, Baik J (2016) Value-cognitive boosting with a support vector machine for cross-project defect prediction. Empir Softw Eng 21(1):43–71

[38] Ryu D, Jang J-I, Baik J (2017) A transfer cost-sensitive boosting approach for cross-project defect prediction. Softw Qual J 25(1):235–272

[39] Yi P, Kou G, Wang G, Wu W, Shi Y (2011) Ensemble of software defect predictors: an ahp-based evaluation method. International Journal of Information Technology & Decision Making 10(01):187–206

[40] Zhang Y, Lo D, Xia X, Sun J (2018) Combined classifier for cross-project defect prediction: an extended empirical study. Frontiers of Computer Science 12(2):280–296

[41] Wang H, Khoshgoftaar TM, Napolitano A (2010) A comparative study of ensemble feature selection techniques for software defect prediction. In: 2010 Ninth international conference on machine learning and applications. IEEE, pp 135–140

[42] Uchigaki S, Uchida S, Toda K, Monden A (2012) An ensemble approach of simple regression models to cross-project fault prediction. In: 2012 13th ACIS International conference on software engineering, artificial intelligence, networking and parallel/distributed computing. IEEE, pp 476–481

[43] Coelho RA, dos RN Guimarães F, Esmin AAA (2014) Applying swarm ensemble

clustering technique for fault prediction using software metrics. In: 2014 13th International conference on machine learning and applications. IEEE, pp 356–361

[44] Li Z, Jing Xiao-Yuan, Zhu X, Zhang H (2017) Heterogeneous defect prediction through multiple kernel learning and ensemble learning. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp 91–102

[45] Tong H, Liu B, Wang S (2019) Kernel spectral embedding transfer ensemble for heterogeneous defect prediction. IEEE Transactions on Software Engineering

[46] Wang T, Li W, Shi H, Liu Z (2011) Software defect prediction based on classifiers ensemble. J Info Comput Sci 8(16):4241–4254

[47] Aljamaan HI, Elish MO (2009) An empirical study of bagging and boosting ensembles for identifying faulty classes in object-oriented software. In: 2009 IEEE Symposium on computational intelligence and data mining. IEEE, pp 187–194

[48] Rathore SS, Kumar S (2017) Linear and non-linear heterogeneous ensemble methods to predict the number of faults in software systems. Knowl.-Based Syst 119:232–256

[49] Ceran, A.A.; Ar, Y.; Tanrıöver, Ö.Ö.; Seyrek Ceran, S. Prediction of software quality with Machine Learning-Based ensemble methods , 4th International Engineering Research Symposium (INERS'22), Volume 81, Part 1, 2023, Pages 18-25, https://doi.org/10.1016/j.matpr.2022.11.229

[50] Matloob, F. et al. Sotware defect prediction using ensemble learning: A systematic literature review. IEEE. Access 9, 98754–98771. https:// doi. org/ 10. 1109/ ACCESS. 2021. 30955 59 (2021)

[51] PandeyS.K. et al. Machine learning based methods for software fault prediction: A survey Expert Systems with Applications. Volume 172, 15 June 2021, 114595

[52] Mehta, S.; Patnaik, K.S. Improved prediction of software defects using ensemble machine learning technique. Neural Comput. Appl. 2021,33, 10551–10562

[53] Balogun, A.O. et al. (2020). SMOTE-Based Homogeneous Ensemble Methods for Software Defect Prediction. In: Gervasi, O., et al. Computational Science and Its Applications – ICCSA 2020. ICCSA 2020. Lecture Notes in Computer Science, vol 12254. Springer, Cham. https://doi.org/10.1007/978-3-030-58817-5_45

[54] Johnson, F., Oluwatobi, O., Folorunso, O. et al. Optimized ensemble machine learning model for software bugs prediction. Innovations Syst Softw Eng 19, 91–101 (2023). https://doi.org/10.1007/s11334-022-00506-x

[55] Shaojian Qiu, Lu Lu, Siyu Jiang, and Yang Guo. An investigation of imbalanced ensemble learning methods for cross-project defect prediction. International Journal of Pattern Recognition and Artificial Intelligence, 33, 01 2019. doi: 10.1142/S0218001419590377.

[56] Manu Banga and Abhay Bansal. Proposed software faults detection using hybrid approach. Security and Privacy, 01 2020. doi: 10.1002/spy2.103.

[57] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. Data quality: Some comments on the nasa software defect datasets. Software Engineering, IEEE Transactions on, 39:1208–1215, 09 2013. doi: 10.1109/TSE.2013.11.

[58] Martin Shepperd, Qinbao SOng, Zhongbin Sun, and Carolyn Mair. Nasa mdp software defects data sets, Mar 2018. URL https://figshare.com/collections/NASA_MDP_Software_Defects_Data_Sets/4054940/1.

[59] Deepti Aggarwal. Software Defect Prediction Dataset. 1 2021. doi: 10. 6084/m9.figshare. 13536506.v1. URL https://figshare.com/articles/dataset/Software_Defect_Prediction_Dataset/13536506.

[60] Saiqa Aleem, Luiz Capretz, and Faheem Ahmed. Benchmarking machine learning techniques for software defect detection. International Journal of Software Engineering & Applications, 6:11–23, 05 2015. doi: 10.5121/ijsea.2015.6302.

[61] Md Fahimuzzman Sohan, Md Alamgir Kabir, Mostafijur Rahman, Touhid Bhuiyan, M. Ismail Jabiullah, and Amarachukwu Felix. Prevalence of Machine Learning Techniques in Software Defect Prediction, pages 257–269. 07 2020. ISBN 978-3- 030-52855-3. doi: 10.1007/978-3-030-52856-0_20.

[62] Ernest Ampomah, Zhiguang Qin, and Gabriel Nyame. Evaluation of tree-based ensemble machine learning models in predicting stock price direction of movement. Information, 11:332, 06 2020. doi: 10.3390/info11060332.

[63] wni Hammouri, Mustafa Hammad, Mohammad Alnabhan, and Fatima Alsarayrah. Software bug prediction using machine learning approach. International Journal of Advanced Computer Science and Applications, 9, 01 2018. doi: 10.14569/IJACSA.2018.090212.

[64] Praman Deep Singh, Anuradha Chug, "Software Defect Prediction Analysis Using Machine Learning Algorithms", 2017, 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence, 775-781, 978-1-5090-3519-9/17 2.

[65] Alsaeedi, M. Z. Khan, "Software Defect Prediction Using Supervised Machine Learning and Ensemble Techniques: A Comparative Study", Journal of Software Engineering and Applications, 2019, 12, 85-100, DOI: 10.4236/jsea.2019.125007

[66] Thota et al, "Survey on software defect prediction techniques", International Journal of Applied Science and Engineering, 17(4), 331–344, https://doi.org/10.6703/IJASE.202012_17(4).331

[67] Shraban Kumar Apat, S V Achuta Rao, P. Santosh Kumar Patra, "Software Bug Prediction Analysis Using Various Machine Learning Approaches", International Journal of Advanced Science and Technology Vol. 29, No. 8s, (2020), pp. 1508-1516

[68] Data Preprocessing in Machine Learning [Steps & Techniques] (v7labs.com)

[69] Rathore, S.S., Kumar, S. An empirical study of ensemble techniques for software fault prediction. Appl Intell 51, 3615–3644 (2021). https://doi.org/10.1007/s10489-020-01935-6