UNIVERSITY OF MISKOLC

FACULTY OF MECHANICAL ENGINEERING AND INFORMATICS

Utilizing Data-Balancing Techniques to Improve AI-Based Prediction of Software Bugs and Code Smells

PhD DISSERTATION

AUTHOR:

Nasraldeen Alnor Adam Khleel
MSc in Software Engineering

József Hatvany Doctoral School of

Information Science, Engineering and Technology

HEAD OF DOCTORAL SCHOOL

Prof. Dr. Jenő SZIGETI

ACADEMIC SUPERVISOR

Dr. Károly Nehéz

Miskolc
2023

Declaration of Authorship

The author hereby declares that this dissertation has not been submitted, either in the same or in a different form, to this or to any other university for obtaining a PhD degree. The author confirms that the submitted work is his own and the appropriate credit has been given where reference has been addressed to the work of others.

Author's declaration

I, the undersigned, Nasraldeen Alnor Adam Khleel, declare that I have prepared this doctoral dissertation and have used only the sources provided.

All parts that I have taken from another source, either directly or in the same content but paraphrased, are clearly marked with the source.

November 20, 2023.

Nasraldeen Alnor Adam Khleel

Acknowledgments

First and foremost, I would like to praise and thank God, Allah, almighty, who has granted me countless blessings, knowledge, inspiration, and opportunity to me so that I was able to accomplish my dissertation.
I would also like to thank the University of Miskolc, Faculty of Mechanical Engineering and Informatics for the opportunity given to me to study for a PhD in Information Technology.

Apart from my efforts to harvest the fruits of this work, the success of this thesis depends on the encouragement and guidelines of many others. This dissertation became a reality because of the help and support of people around me; it was also a result of a lot of effort and hard work during the past four years.

Above all, I would like to thank my supervisor, Dr. Károly Nehéz, for his continued to support, direction, and encouragement over the past years; this dissertation would not have been possible without him, I would also like to thank everyone in the computer science department.

Furthermore, I am grateful to many of my colleagues in our department for their help and support in organizing the events associated with the doctorate. I also express my thankful feelings to my colleagues for their help. Lastly but not least, I owe my loving thanks and deep sense of gratitude to my siblings and my parents for their support and encouragement.

Nasraldeen Alnor Adam Khleel

Table of Contents

Declaration of Authorship I
Author's declaration I
Acknowledgments II
List of Abbreviations VI
List of Figures VIII
List of Tables XI
Chapter 1 Introduction 1
1.1 Motivation 2
1.2 Problem Statement 3
1.3 The objectives of the thesis 3
1.4 Dissertation Guide 4
Chapter 2 Literature Review and Theoretical Background 5
2.1 Software Bugs 5
2.1.1 Software Bug Prediction (SBP) 5
2.1.2 Software Bug Prediction Approaches 6
2.1.2.1 With-in Project Defect Prediction (WPDP) 6
2.1.2.2 Cross Project Defect Prediction (CPDP) for Similar Dataset 6
2.1.2.3 Cross Project Defect Prediction (CPDP) for Heterogeneous Dataset 6
2.2 Code Smells 7
2.2.1 Types of Code Smells 7
2.2.1.1 God class 7
2.2.1.2 Data class 7
2.2.1.3 Feature envy 7
2.2.1.4 Long method 8
2.2.2 Code Smells Detection 8
2.3 Software Metrics 8
2.4 Summary 11
Chapter 3 Artificial Intelligence (AI) 12
3.1 Artificial Intelligence Techniques 12
3.1.1 Machine Learning (ML) 12
3.1.1.1 Supervised learning 13
3.1.1.2 Unsupervised learning 15
3.1.1.3 Reinforcement learning 16
3.1.2 Artificial Neural Networks (ANNs) 16
3.1.2.1 Multi-layer Perceptron (MLP) 17
3.1.2.2 Deep learning (DL) 17
3.1.2.3 Recurrent Neural Networks (RNNs) 19
3.2 Summary 22
Chapter 4 Data Imbalance and Data-Balancing Methods 23

4.1 Data Imbalance 23

4.2 Data-Balancing Methods 23

4.2.1 Data Sampling (Resampling) Methods 24

4.2.1.1 Undersampling Methods 24

4.2.1.2 Oversampling Methods 24

4.2.1.3 Hybrid (Combined-Sampling Methods) 25

4.3 Summary 25

Chapter 5 Proposed Methodology and Implementation 27

5.1 Experimental Design 27

5.1.1 Proposed Approaches 27

5.1.2 The Public Benchmark Datasets Used in This Research 28

5.1.2.1 Software Bug Data Sets 28

5.1.2.2 Code Smells Data Sets 29

5.1.3 Data Pre-processing 29

5.1.4 Features Selection 30

5.1.5 Balancing Data sets 32

5.1.6 Models Building and Evaluation 36

5.2 Summary 40

Chapter 6 Experimental Results and Discussion of Software Bugs Prediction (SBP) 41

6.1 ML Techniques in SBP 41

6.2 LSTM and GRU with Undersampling Methods in SBP 45

6.3 Bi-LSTM with Oversampling Methods in Software Defect Prediction (SDP) 51

6.4 CNN and GRU with Hybrid (combined)-Sampling Methods in SDP 61

6.5 Summary 72

Chapter 7 Experimental Results and Discussion of Code Smells Detection 73

7.1 ML techniques with Oversampling Methods in Code Smells Detection 73

7.2 A Convolutional Neural Network (CNN) with Oversampling Methods 79

7.3 Bi-LSTM and GRU with Under and Oversampling Methods in Code Smells Detection 84

7.4 Summary 95

Chapter 8 Conclusion 96

8.1 Contributions 96

8.1.1 Theses - New Scientific Results 96

8.2 Future Research Direction 99

Appendices 100

Appendix 1: LSTM and GRU with Undersampling Methods in SBP 100

Appendix 2: Bi-LSTM with Oversampling Methods in SDP 101

Appendix 3: CNN and GRU with Hybrid (Combined)-Sampling Methods in SDP 102

Appendix 4: Bi-LSTM and GRU with Under and Oversampling Methods 104

Author's Publication 111

Publications Related to the Dissertation 111

Other Publications Journal Articles and Conference Proceeding 112

References ____ 113

List of Abbreviations

(SBP)	Software Bugs Prediction
(AI)	Artificial Intelligence
(ML)	Machine Learning
(ANNs)	Artificial Neural Networks
(CI)	Continuous Integration
(CD)	Continuous Deployment
(WPDP)	With-in Project Defect Prediction
(CPDP)	Cross Project Defect Prediction
(LOC)	Lines of Code
(CCN)	Cyclomatic Complexity Number
(DIT)	Depth of Inheritance Tree
(CBO)	Coupling Between Objects
(NOC)	Number of Children
(RFC)	Response for a Class
(CYCLO)	McCabe's CYCLOmatic complexity
(LCOM)	Lack of Cohesion between Methods
(CLASS_FAN_OUT)	Class Fan Out Complexity
(LAA)	Locality of Attribute Accesses
(LOCNAMM*)	Lines of Code Excluding Accessor and Mutator Methods
(WMC)	Weighted Methods per Class
(TCC)	Tight Class Cohesion
(ATFD)	Access To Foreign Data
(NOAM)	Number of Accessor Methods
(NOM)	Number of Methods
(WMCNAMM*)	Weighted Methods Count of Not Accessor or Mutator Methods
(FDP)	Foreign Data Providers
(NOPA)	Number of Public Attributes
(NOPK)	Number of Packages
(AMWNAMM*)	Average Methods Weight of Not Accessor or Mutator Methods
(NMO)	Number of Methods Overridden
(NOCS)	Number of Classes
(AMW)	Average Methods Weight
(CFNAMM*)	Called Foreign Not Accessor or Mutator Methods
(NIM)	Number of Inherited Methods
(NOMNAMM*)	Number of Methods Excluding Accessor and Mutator Methods
(MAXNESTING)	Maximum Nesting Level of Control Structures
(CINT)	Coupling Intensity
(NOII)	Number of Implemented Interfaces
(NOA)	Number of Attributes
(WOC)	Weight of Class
(CDISP)	Coupling Dispersion
(CLNAMM)	Called Local Not Accessor or Mutator Methods
(MaMCL§)	Maximum Message Chain Length
(NOP)	Number of Parameters
(MeMCL§)	Mean Message Chain Length
(NOAV)	Number of Accessd Variables
(NMCS§)	Number of Message Chain Statements
(ATLD*)	Access To Local Data

(CC)	Control Coupling
(NOLV)	Number of Local Variable
(CM)	Number of Methods Affected by the Measured Method
(DT)	Decision Tree
(ID3)	Iterative Dichotomiser 3
(CART)	Classification and Regression Trees
(RF)	Random Forest
(NB)	Naïve Bayes
(SVM)	Support Vector Machine
(KNN)	K-Nearest Neighbor
(LR)	Logistic Regression
(XGB)	XGBoost
(MLP)	Multi-layer Perceptron
(DL)	Deep Learning
(Relu)	Rectified Linear unit
(Tanh)	Hyperbolic Tangent
(CNN)	Convolutional Neural Network
(RNNs)	Recurrent Neural Networks
(LSTM)	Long-Short-Term-Memory
(Bi-LSTM)	Bidirectional Long-Short-Term-Memory
(GRU)	Gated Recurrent Unit
(SMOTE)	Synthetic Minority Oversampling Technique
(QC)	Qualitas Corpus
(MCC)	Matthews Correlation Coefficient
(ROC)	Receiver Operating Characteristic
(AUC)	Area Under the ROC Curve
(AUCPR)	The Area Under the Precision-Recall Curve
(MSE)	Mean Square Error
(TPR)	True Positive Rate
(FPR)	False Positive Rate
(TNR)	True Negative Rate
(FNR)	False Negative Rate
(SDP)	Software Defect Prediction

List of Figures

Figure 3.1 The typical ANN architecture[70] ... 17
Figure 3.2 The typical CNN architecture[80] ... 18
Figure 3.3 Interacting layers of the repeating module in an LSTM Networks[40] ... 20
Figure 3.4 Interacting layers of the repeating module in a Bi-LSTM Network[86] ... 21
Figure 3.5 Interacting layers of the repeating module in a GRU Networks[7] ... 22
Figure 4.1 Shows how data sampling methods deal with class imbalance ... 25
Figure 5.1 The architecture of the methodology followed in the dissertation ... 27
Figure 5.2 Distribution of learning instances over the original and balanced data sets (The public unified bug dataset)-by applying the Near Miss method ... 34
Figure 5.3 Distribution of learning instances over the original and balanced data sets (The PROMISE datasets)-by applying the SMOTE Tomek method ... 34
Figure 5.4 Distribution of learning instances over the original and balanced data sets (The PROMISE datasets)-by applying the Random Oversampling and SMOTE methods ... 35
Figure 5.5 Distribution of learning instances over the original and balanced data sets (The Qualitas Corpus Systems)-by applying the SMOTE method ... 35
Figure 5.6 Distribution of learning instances over the original and balanced data sets (The Qualitas Corpus Systems)-by applying the Random Oversampling method ... 36
Figure 5.7 Distribution of learning instances over the original and balanced data sets (The Qualitas Corpus Systems)-by applying the Random Oversampling and Tomek Links methods ... 36
Figure 6.1 Comparison of ROC curves for Models Across the jm1 Dataset ... 42
Figure 6.2 Comparison of ROC curves for Models Across the pc1 Dataset ... 42
Figure 6.3 Comparison of ROC curves for Models Across the kc1 Dataset ... 43
Figure 6.4 Comparison of ROC curves for Models Across the kc2 Dataset ... 43
Figure 6.5 Showcases the boxplots illustrating the performance measures achieved by the proposed models on all datasets, encompassing both class-level and file-level metrics ... 46
Figure 6.6 Represents the training and validation accuracy of the models across all datasets - class-level metrics ... 47
Figure 6.7 Represents the training and validation accuracy of the models across all datasets - file-level metrics ... 47
Figure 6.8 Represents the training and validation loss of the models across all datasets - class-level metrics ... 48
Figure 6.9 Represents the training and validation loss of the models across all datasets - file-level metrics ... 48
Figure 6.10 Illustrates the ROC Curves of the models across all datasets - class-level metrics ... 49
Figure 6.11 Illustrates the ROC Curves of the models across all datasets - file-level metrics ... 49
Figure 6.12 Boxplots represent performance measures obtained by the model on the original and balanced datasets ... 54
Figure 6.13 Training and validation accuracy for the original datasets ... 55
Figure 6.14 Training and validation accuracy for the balanced datasets - Random Oversampling ... 55
Figure 6.15 Training and validation accuracy for the balanced datasets – SMOTE ... 56
Figure 6.16 Training and validation loss for the original datasets ... 56
Figure 6.17 Training and validation loss for the balanced datasets - Random Oversampling ... 57
Figure 6.18 Training and validation loss for the balanced datasets - SMOTE ... 57
Figure 6.19 ROC curves for the original datasets ... 58
Figure 6.20 ROC curves for the balanced datasets- Random Oversampling ... 58
Figure 6.21 ROC curves for the balanced datasets- SMOTE ... 59
Figure 6.22 Boxplots represent performance measures obtained by proposed models on all datasets ... 64

Figure 6.23 Training and Validation Accuracy for the original data sets - CNN model 65
Figure 6.24 Training and Validation Accuracy for the balanced data sets - CNN model..................... 65
Figure 6.25 Training and Validation Accuracy for the original data sets - GRU model 66
Figure 6.26 Training and Validation Accuracy for the balanced data sets - GRU model..................... 66
Figure 6.27 Training and Validation Loss for the original data sets - CNN model 67
Figure 6.28 Training and Validation Loss for the balanced data sets - CNN model 67
Figure 6.29 Training and Validation Loss for the original data sets - GRU model 68
Figure 6.30 Training and Validation Loss for the balanced data sets - GRU model 68
Figure 6.31 ROC curves for the original data sets - CNN model 69
Figure 6.32 ROC curves for the balanced data sets - CNN model 69
Figure 6.33 ROC curves for the original data sets - GRU model 70
Figure 6.34 ROC curves for the balanced data sets - GRU model 70
Figure 7.1 Box Plots represent the models' performance measures on all considered code smells_ original datasets 76
Figure 7.2 Box Plots represent the models' performance measures on all considered code smells_ balanced datasets.. 76
Figure 7.3 The ROC curves obtained by the models on all considered code smells_ original datasets 77
Figure 7.4 The ROC curves obtained by the models on all considered code smells_ balanced datasets 78
Figure 7.5 Boxplots represent performance measures obtained by CNN Model.................................. 81
Figure 7.6 Training and Validation Accuracy over original datasets 81
Figure 7.7 Training and Validation Accuracy over balanced datasets.. 82
Figure 7.8 Training and validation loss over original datasets 82
Figure 7.9 Training and validation loss over balanced datasets... 83
Figure 7.10 Training and Validation Accuracy on the original datasets using Bi-LSTM Model......... 87
Figure 7.11 Training and Validation Accuracy on the original datasets using GRU Model 87
Figure 7.12 Training and Validation Loss on the original datasets using Bi-LSTM Model................. 88
Figure 7.13 Training and Validation Loss on the original datasets using GRU Model 88
Figure 7.14 ROC curves for the original datasets - Bi-LSTM Model 89
Figure 7.15 ROC curves for the original datasets - GRU Model.. 89
Figure 7.16 Boxplots representing performance measures obtained by models on the original datasets 90
Figure 7.17 Boxplots representing performance measures obtained by models on the balanced datasets- Random Oversampling 91
Figure 7.18 Boxplots representing performance measures obtained by models on the balanced datasets- Tomek links 92

Appendix 1: 0.1 Figure 1. Illustrates the AUCPR of the models across all datasets - class-level metrics 100
Appendix 1: 0.2 Figure 2. Illustrates the AUCPR of the models across all datasets - file-level metrics 100

Appendix 2: 0.1 Figure 1. AUCPR for the original datasets 101
Appendix 2: 0.2 Figure 2. AUCPR for the balanced datasets - Random Oversampling 101
Appendix 2: 0.3 Figure 3. AUCPR for the balanced datasets – SMOTE 102

Appendix 3: 0.1 Figure 1. AUCPR for the original data sets - CNN model 102
Appendix 3: 0.2 Figure 2. AUCPR for the balanced data sets - CNN model........................ 103

Appendix 3: 0.3 Figure 3. AUCPR for the original data sets - GRU model 103
Appendix 3: 0.4 Figure 4. AUCPR for the balanced data sets - GRU model 104

Appendix 4: 0.1 Figure 1. Training and Validation Accuracy on the balanced datasets using Bi-LSTM Model-Random Oversampling 104
Appendix 4: 0.2 Figure 2. Training and Validation Accuracy on the balanced datasets using Bi-LSTM Model- Tomek links 105
Appendix 4: 0.3 Figure 3. Training and Validation Accuracy on the balanced datasets using GRU Model-Random Oversampling 105
Appendix 4: 0.4 Figure 4. Training and Validation Accuracy on the balanced datasets using GRU Model- Tomek links 106
Appendix 4: 0.5 Figure 5. Training and Validation Loss on the balanced datasets using Bi-LSTM Model-Random Oversampling 106
Appendix 4: 0.6 Figure 6. Training and Validation Loss on the balanced datasets using Bi-LSTM Model- Tomek links 107
Appendix 4: 0.7 Figure 7. Training and Validation Loss on the balanced datasets using GRU Model-Random Oversampling 107
Appendix 4: 0.8 Figure 8. Training and Validation Loss on the balanced datasets using GRU Model - Tomek links 108
Appendix 4: 0.9 Figure 9. ROC curves for the balanced datasets - Bi-LSTM Model-Random Oversampling 108
Appendix 4: 0.10 Figure 10. ROC curves for the balanced datasets - Bi-LSTM Model- Tomek links 109
Appendix 4: 0.11 Figure 11. ROC curves for the balanced datasets - GRU Model-Random Oversampling 109
Appendix 4: 0.12 Figure 12. ROC curves for the balanced datasets - GRU Model- Tomek links 110

List of Tables

Table 2.1 Show the static code metrics 9
Table 2.2 Description list of 20 traditional static code metrics 10
Table 2.3 Descriptions of McCabe's and Halstead Metrics 10
Table 5.1 Description of the NASA datasets 28
Table 5.2 Description of the public unified bug dataset 29
Table 5.3 Description of the PROMISE datasets 29
Table 5.4 Description of the Qualitas Corpus Systems 29
Table 5.5 Parameter settings of the models (Classical techniques) 38
Table 5.6 Parameter settings of the models (Advanced techniques) 38
Table 5.7 Confusion matrix 38
Table 6.1 Performance measures of the proposed models on the jm1 dataset 41
Table 6.2 Performance measures of the proposed models on the pc1 dataset 41
Table 6.3 Performance measures of the proposed models on the kc1 dataset 41
Table 6.4 Performance measures of the proposed models on the kc2 dataset 41
Table 6.5 Comparing the results of our study with the results of studies that used the same dataset and algorithms across the jm1 and pc1 dataset 43
Table 6.6 Comparing the results of our study with the results of studies that used the same dataset and algorithms across the kc1 and kc2 datasets 44
Table 6.7 Performance measures for the proposed models over class level metrics dataset ... 45
Table 6.8 Performance measures for the proposed models over file level metrics dataset 46
Table 6.9 Comparison of the proposed approach with other existing approaches based on the accuracy and AUC 50
Table 6.10 Performance analysis for proposed Bi-LSTM Network - Original Datasets 51
Table 6.11 Performance analysis for proposed Bi-LSTM Network - Balanced Datasets using Random Oversampling Technique 52
Table 6.12 Performance analysis for proposed Bi-LSTM Network - Balanced Datasets using SMOTE Technique 53
Table 6.13 Comparison of the results of the proposed Bi-LSTM Model based on the original and balanced datasets in terms of accuracy using paired t-test 53
Table 6.14 Performance measures of the baseline model (RF) and Bi-LSTM 59
Table 6.15 Comparison of the proposed Bi-LSTM with other existing approaches 60
Table 6.16 Performance analysis for proposed CNN Model-Original Data sets 61
Table 6.17 Performance analysis for proposed CNN Model-Balanced Datasets 61
Table 6.18 Performance analysis for proposed GRU Model-Original Data sets 62
Table 6.19 Performance analysis for proposed GRU Model-Balanced Datasets 62
Table 6.20 Performance analysis for proposed models based on precision and recall measures - CNN Model 62
Table 6.21 Performance analysis for proposed models based on precision and recall measures - GRU Model 62
Table 6.22 Summarizes the range of measures values for the proposed models on the original and balanced datasets 63
Table 6.23 Comparison of the proposed models in terms of accuracy using paired t-test 64
Table 6.24 Performance measures of the baseline model (RF) and proposed models 71
Table 6.25 Comparison of the proposed models with other existing approaches 71

Table 7.1 Evaluation Results for the Class-Level Dataset: God class_ original and balanced datasets....74
Table 7.2 Evaluation Results for the Class-Level Dataset: Data class_ original and balanced datasets....74
Table 7.3 Evaluation Results for the Method-Level Dataset: Long method_ original and balanced datasets....75
Table 7.4 Evaluation Results for the Method-Level Dataset: Feature envy_ original and balanced datasets....75
Table 7.5 Comparison of the proposed method with other existing methods based on the accuracy....78
Table 7.6 Comparison of the proposed method with other existing methods based on AUC....79
Table 7.7 Performance analysis for proposed CNN Model - Original Datasets....80
Table 7.8 Performance analysis for proposed CNN Model - Balanced Datasets....80
Table 7.9 Comparison of the proposed method with other existing methods based on the accuracy....83
Table 7.10 Evaluation results for the original datasets....84
Table 7.11 Evaluation results for the balanced datasets - Random Oversampling....85
Table 7.12 Evaluation results for the balanced datasets - Tomek links....86
Table 7.13 Comparison of the proposed models in terms of accuracy using paired t-test- based on the original and balanced datasets (using Random Oversampling)....92
Table 7.14 Comparison of the proposed models in terms of accuracy using paired t-test- based on the original and balanced datasets (using Tomek Links)....93
Table 7.15 Comparison of the proposed models with other existing approaches based on the accuracy....93
Table 7.16 Comparison of the proposed models with other existing approaches based on AUC....94
Table 7.17 Comparison of the proposed models with other existing approaches in terms of accuracy averages using paired t-test- based on Random Oversampling....94
Table 7.18 Comparison of the proposed models with other existing approaches in terms of accuracy averages using paired t-test- based on Tomek Links....95

Chapter 1 Introduction

In the field of software engineering, ensuring the quality of software systems is of paramount importance. Software quality assurance is a crucial discipline within software engineering that focuses on ensuring the high standards, reliability, and functionality of software products throughout their development life cycle. The primary goal of software quality assurance is to identify and mitigate defects, errors, code smells and inconsistencies in software, ultimately leading to the delivery of a high-quality product that meets user requirements and expectations [1]. Due to the increasing size and complexity of software products and inadequate software testing, no system or software can claim to be free of software bugs or code smells. Software bugs and code smells can significantly impact software applications' performance, maintainability, and user experience. Detecting and predicting these issues early in the software development life cycle can save substantial time, effort, and resources. There are many activities related to software testing, such as implementing processes, procedures, and standards that must be carried out in a specific sequence to ensure that quality objectives are achieved or testing a product for issues such as software bugs and code smells. Software bugs are defects or errors in computer programs or systems that cause incorrect or unexpected operations that negatively affect software quality, reliability, and maintenance costs [2]. Software Bugs Prediction (SBP) is one of the most popular and active research areas in software engineering. SBP is a process for classifying fault-prone software modules based on some underlying properties of the systems, like software metrics that are extracted and collected from real data sets (historical data) during the software development process [3]. Code smells are one of the most accepted approaches to identifying design problems in the source code, which refers to any symptom or anomaly in the source code that violates design or implementation principles. The detection of code smells is a particularly crucial step for guiding the subsequent steps in the refactoring process. Early detection of code smells is vital to aid software maintainability and improve software quality [4]. Software metrics have essential roles in predicting software bugs and code smells, and most recent strategies for predicting software bugs and code smells rely on software metrics as independent variables. Software metrics are essential aids in measuring and improving software quality, and these metrics are used to measure and characterize software engineering products[5]. The critical role of software metrics is to estimate and measure some characteristics of systems, such as classes, inheritance, encapsulation, etc.[6]. The most popular software metrics are object-oriented metrics, which have been presented by Abreu, Chidamber and Kemerer, Li and Henry, MOOD, Lorenz, and Kidd. These metrics can be classified into different classes, like metrics for source code analysis, software testing, quality assurances, etc.[4]. Static code analysis is a method of analyzing source code without its execution to find potential problems like software bugs and code smells that might arise at runtime. So, static code analysis aims to check the quality of the source code and address weaknesses[7]. Based on the literature review. Recently, many commercial and open-source tools evolved for static code analysis to provide an efficient, value-added solution to many of the problems that software development organizations face. However, numerous false positives and negative results make these tools hard to use in practice[8]. So, another methodology or approach for static code analysis must be found, such as artificial intelligence techniques. Artificial Intelligence (AI) is a wide-ranging branch of computer science concerned with the simulation of human intelligence in machines that are programmed to think like humans and mimic their actions. AI handles issues related to implementing human behaviour and emotion and software intelligence. The most popular AI

techniques used for the prediction of software bugs and code smells are Machine Learning (ML) techniques. The ML field is developed from the expanded field of AI, which aims to imitate human intelligence abilities by machines. ML is the process of gaining knowledge from historical data. ML uses statistical rules to build various mathematical models for creating the conclusion from the data sample[9]. ML is an area of research where computer programs can learn and get better at performing specific tasks by training on massive quantities of historical data. ML algorithms can be applied to analyze data from different perspectives to allow developers to obtain helpful information[4]. ML techniques, and software metrics have emerged as powerful tools for automating the prediction of software bugs and code smells[5]. However, one major challenge faced in this domain is the class imbalance problem, where the distribution of classes in the training dataset is uneven. In other words, one class has significantly more instances than the others, leading to an imbalanced representation of classes. The class imbalance issue poses a significant obstacle as it can lead to biased models that fail to accurately capture the rare occurrences of software bugs or code smells, thus affecting the overall predictive performance[7]. Therefore, this research aims to explore the role of data-balancing methods in addressing the class imbalance problem when applying ML techniques for predicting software bugs and code smells using software metrics. The research will begin with a comprehensive literature review, examining existing studies predicting software bugs and code smells using ML techniques. This review will also encompass different data-balancing methods commonly employed in the field. The research outcomes will provide valuable insights and guidelines for software developers and researchers aiming to leverage ML-based techniques to accurately predict software bugs and code smells. In conclusion, this dissertation aims to contribute to the field of software engineering by investigating the application of data-balancing methods in ML-based prediction of software bugs and code smells using software metrics. By addressing the class imbalance problem, the research endeavours to enhance the accuracy and reliability of predictive models, ultimately assisting in developing more robust and high-quality software systems[10].

1.1 Motivation

The software industry plays a critical role in today's technologically advanced world, with software systems powering various aspects of our lives. However, software bugs and code smells can lead to system failures, security vulnerabilities, and compromised user experiences. Identifying software bugs and code smells is usually a challenging task due to the huge code base of software projects, and developers spend a significant amount of time locating and fixing them, making this an active research area in software engineering. To produce high-quality software and gain customer loyalty, the final product should have as few defects as possible[11]. Detecting and addressing these issues early in the software development process is essential to ensure reliable and high-quality software systems. ML techniques, and software metrics have shown promise in automating the prediction of software bugs and code smells. However, the class imbalance problem remains a significant challenge in this domain, affecting the accuracy and effectiveness of the predictive models[7]. Therefore, the motivation behind this dissertation is driven by the need to address the class imbalance problem in the ML-based prediction of software bugs and code smells using software metrics and shed light on the suitability and effectiveness of various data-balancing methods commonly employed in the domain of the prediction of software bugs and code smells. By investigating and evaluating data-balancing methods, this research seeks to improve the accuracy and reliability of

predictive models, ultimately contributing to developing more robust and high-quality software systems.

1.2 Problem Statement

Software bugs and code smells can be identified by manual or automated source code analysis. The manual recognition of software bugs and code smells on the source code by developers is an error-prone, costly, and time-consuming activity since it depends on the developer's degree of experience and perception[12]. Previous work provided several tools for predicting software bugs and code smells. These tools rely on prediction rules that compare the values of relevant software metrics extracted from source code against empirically identified thresholds to discriminate defective source code. The limitations of these tools are that the performance is strongly influenced by the thresholds needed to identify defective and non-defective instances. To overcome these limitations, researchers recently adopted and developed many automatic tools, such as machine-learning techniques, where a classifier is trained on previous source code releases by exploiting a set of independent variables (e.g., structural, historical, or textual metrics). But recent studies indicate that machine-learning techniques are not always suitable for predicting software bugs and code smells due to the problem of imbalanced data[13]. The data sets of software bugs and code smells are often imbalanced, which means the defective modules are often less than the non-defective ones. Using an imbalanced data set to train classification algorithms can lead to misclassification, as the classifier may be biased and not correctly classify instances of the minority label. The problem addressed by this dissertation is the lack of effective approaches to address the class imbalance problem in the ML-based prediction of software bugs and code smells using software metrics. Existing research in this area often overlooks the impact of class imbalance on model performance and fails to provide comprehensive solutions. As a result, the accuracy and reliability of the predictive models are compromised, leading to suboptimal detection of software bugs and code smells in real-world software projects. The inadequate handling of class imbalance in software bugs and code smell prediction can have severe consequences[14]. Most ML techniques can predict better when the number of instances of each class is equal. So, data imbalance is the biggest problem faced by ML techniques. This problem severely hinders the efficiency of these techniques and produces imbalanced false-positive and false-negative results. False negatives, where actual software bugs or code smells are incorrectly classified as non-issues, can result in software systems with hidden vulnerabilities or quality issues. False positives, where non-issues are incorrectly classified as software bugs or code smells, can lead to wasted development efforts and unnecessary maintenance activities. To address this problem, this dissertation aims to investigate and evaluate various data-balancing methods in the context of ML-based prediction of software bugs and code smells using software metrics. The research seeks to identify and employ suitable data-balancing techniques that effectively address the class imbalance problem, improve model sensitivity to the minority class, and enhance the accuracy and reliability of the predictive models[7], [15].

1.3 The objectives of the thesis

ML techniques and data-balancing methods can provide new and performing ways for software bug and code smell prediction, with more flexibility than heuristics approaches, and can also help software companies to reduce rework and improve the quality and reliability of software. To the best of our knowledge, based on the literature review, no more research is conducted to

predict software bugs and code smells using ML techniques combined with data-balancing methods. Based on the previous studies, balancing the data by applying data-balancing methods can improve the performance of ML models in predicting software bugs and code smells. The specific objectives of this thesis are:

- To investigate the standard machine-learning techniques used for predicting software bugs and code smells.
- To assess the impact of class imbalance on the performance of ML-based prediction models for software bugs and code smells. This involves analyzing the biases introduced by class imbalance and understanding how they affect the predictive models' accuracy.
- To evaluate various data-balancing methods to address class imbalance in software bug and code smell prediction.
- To enhance the performance of predictive models for software bugs and code smells by developing a novel prediction methodology based on machine-learning techniques combined with data-balancing methods. I will apply various machine-learning algorithms and data-balancing methods to develop the methodology.
- To validate the effectiveness of the developed methodology and the impact of data-balancing methods using real-world software datasets. The validation will involve conducting several experiments and comparisons with baseline models, evaluating the performance measures, and assessing the statistical significance of the results.
- To show that the performance of machine-learning techniques in predicting software bugs and code smells can be significantly improved when balancing the data set by applying data-balancing methods.

1.4 Dissertation Guide

The remaining structure of this dissertation is organized as follows. Chapter 2 presents a theoretical background, and the literature is addressed based on the software bugs, code smells, and software metrics. Chapter 3 provides an overview of artificial intelligence techniques. Specifically, it describes the artificial intelligence techniques used in this research work such as ML and Artificial Neural Networks (ANNs). Chapter 4 provides a short background of imbalanced data and data-balancing methods. Chapter 5 presents the proposed methodology and implementation, which describes the experiments performed. Several experiments are conducted to predict software bugs and code smells based on ML techniques and data-balancing methods. Chapter 6 presents the experimental results and discussion of SBP, describing the experiment outcome and discussion. Chapter 7 presents the experimental results and discussion of code smell detection, which describes the experiments outcome and discussion. Chapter 8 presents the conclusion, firstly, contributions involving new scientific results are presented, and then the future research direction is presented.

Chapter 2 Literature Review and Theoretical Background

This chapter addresses the theoretical background and literature related to software bugs, code smells, and software metrics. This comprehensive exploration delves into the fundamental concepts and theories surrounding software defects, identifying code smells, and the software metrics used to quantify and assess software quality. By examining the existing body of knowledge, this chapter establishes a solid foundation for the subsequent analysis and research conducted in this field. Furthermore, this chapter also discusses the public benchmark datasets of software bugs and code smells. These datasets, meticulously curated and made accessible to researchers and practitioners, serve as valuable resources for evaluating and comparing various bug detection and code smell detection techniques. The availability of these standardized datasets fosters reproducibility and facilitates advancements in bug detection methodologies, ultimately contributing to the ongoing improvement of software reliability and maintainability.

2.1 Software Bugs

Due to the expansion in the scale of software projects and the increase in complexity, software bug prediction has become the focus of attention to increase software quality[16], [17]. Software bugs can be defined as defects or faults in computer programs that occur during the software development process which may cause many problems for users and developers aside and may lead to the failure of the software to meet the desired expectations, and reduce customer satisfaction[18], [19]. Software bugs identify are one of the most common causes of wasted time and increase maintenance costs during the software lifecycle. Where early prediction of software bugs in the early stages of software development can improve the quality and reliability of systems, and reduce development costs, time, rework efforts, etc.[11]. Dealing with software bugs during the development process is problematic, as critical software bugs lead to potential risks that can lead to project failure. To produce high-quality software, the final product delivered should have as limited software bugs as possible[20]. The software bugs are classified into two classes: intrinsic software bugs refer to bugs that were introduced by one or more specific changes to the source code and extrinsic software bugs refer to bugs that were introduced by changes not recorded in the version control system[21]. Developers employ various techniques like debugging tools, code reviews, unit testing, and system testing to detect and resolve software bugs before releasing software to users. In recent years, the adoption of agile development methodologies and continuous integration/continuous deployment (CI/CD) practices has helped in catching software bugs early and reducing their impact. Additionally, bug bounty programs, where individuals are rewarded for discovering and reporting vulnerabilities, have gained popularity in promoting proactive bug detection. Despite advancements in bug detection and prevention, software bugs can never be eliminated. The complexity of modern software systems and the constant evolution of technology make bug-free software an elusive goal. However, with vigilant testing, thorough debugging, and continuous improvement practices, developers can minimize the occurrence and impact of software bugs, resulting in more reliable and secure software products[22].

2.1.1 Software Bug Prediction (SBP)

Predicting software bugs helps in improving the overall quality and reliability of the software. By identifying potential issues in advance, developers can implement preventive measures, conduct targeted testing, and ensure that the software meets the required quality standards[18].

Moreover, predicting software bugs is not only about preventing immediate issues but also about continuously improving the software development process. By analyzing past bug data and patterns, developers can identify areas of weakness, improve coding practices, enhance testing strategies, and implement measures to prevent similar software bugs in future projects[19]. SBP is a mechanism that can be used to trace modules in software and determines whether a software module is faulty by considering some characteristics of parameters collected from software projects[23]. The process of SBP refers to the techniques or tools that use historical defect data to classify defect-prone software modules and build a relationship between software metrics and software defects. The SBP process depends on three main components: dependent variables, independent variables, and a model. Dependent variables are the defect data for the piece of code (defective or non-defective), which can be binary or ordinal variables. Independent variables (inputs) are the software metrics that score the software code. The model contains the rules or algorithms which predict the dependent variable from the independent variables[24]. The studies' efforts in building SBP models can be categorized into two approaches: the first approach is to manually design new features or new sets of features to represent defects, while the second approach involves applying new and improved ML-based classifiers. Current work in predicting software bugs focuses on the second approach that includes: estimating the number of defects in software systems, discovering how software defects relate to software metrics and classifying software defects into two categories of "defect-prone and non-defect-prone"[16].

2.1.2 Software Bug Prediction Approaches

Based on the type of data and the context of the prediction, SBP can be categorized into different types, which are:

2.1.2.1 With-in Project Defect Prediction (WPDP)

The With-in Project Defect Prediction (WPDP) approach involves using historical data to predict defects within a single project. WPDP approach uses data from the same project to train the prediction models, such as source code metrics, bug reports, and code reviews. This approach is usually more accurate since it is based on the specific context of the predicted project, but it requires a significant amount of historical data from the same project[25].

2.1.2.2 Cross Project Defect Prediction (CPDP) for Similar Dataset

Cross Project Defect Prediction (CPDP) approach for a similar dataset: This approach involves predicting defects in a new project using historical data from similar projects. The CPDP approach uses data from one or more similar projects to train the prediction models and then apply them to the new project. This approach can be useful when there is not enough data for WPDP. Still, it assumes that the new project has a similar development context to the projects used for training[25].

2.1.2.3 Cross Project Defect Prediction (CPDP) for Heterogeneous Dataset

Cross Project Defect Prediction (CPDP) approach for a heterogeneous dataset: This approach involves predicting defects in a new project using historical data from projects that differ in their development context or characteristics. The CPDP approach uses data from one or more heterogeneous projects to train the prediction models and then apply them to the new project.

This approach can be challenging since the development contexts of the projects used for training and the new project may differ significantly. Still, it can be useful when there is insufficient data for WPDP or CPDP for a similar dataset[25].

2.2 Code Smells

Code smells are design issues or changes to source codes because of activities performed by developers during emergencies or coding solutions that indicate a violation of software design rules, e.g.: abstraction or hierarchy encapsulation which can cause serious problems during systems maintenance and may impact the software quality in the future[26], [27]. Code smells may lead to future degradation in software projects making software hard to evolve and maintain, and it can effective indicate whether source code should be refactored [28], [29] Code smells are often associated with potential software bugs or vulnerabilities. They can indicate areas of code that are more prone to errors, such as complex conditional logic, unhandled exceptions, or inconsistent naming conventions. By detecting code smells, developers can proactively address these areas, reducing the likelihood of software bugs and improving the overall reliability and robustness of the software[30].

2.2.1 Types of Code Smells

There are many types of code smells but the most common are God class, Data class, Feature envy, and Long method.

2.2.1.1 God class

God classes refer to large, complex, and non-cohesive modules or classes that violate the principle of implementing only one concept per class and dominate a significant part of the main system behaviour by implementing almost all the system functionalities[28]. It is distinguished by its complexity and encompassing many instance variables and methods [19], [31].

2.2.1.2 Data class

Data Class is a class that has only data without functions or any behaviors and does not process this data[13], [28], [32].Or it is a class that passively stores data[33]. This class constitutes smells that contain something unnecessary whose removal can make code easier to understand, effective, and cleaner[34].

2.2.1.3 Feature envy

Feature Envy is a sign of a breach of the rule of grouping behaviour with related data and happens when a method is more interested in other properties of the classes than in the ones from its class[35]. This kind of smell affects the coupling, cohesion, and encapsulation design aspects of the system, representing a problem in the abstract design of the system. It is classified as a coupler smell and affects method/property entities. Thus, this method tends to make so many calls to use the data of the other classes [28], [34].

2.2.1.4 Long method

The Long Method code smells refer to the method that is too long and increases the system's compatibility. It is classified as a blotter smell that affects method-level entities[35]. It is methods that tend to centralize a class's functionality and tends to have too much code, to be complex, to be difficult to understand, and to use large amounts of data from other classes [4], [36].

2.2.2 Code Smells Detection

Code smell detection is fundamental to improving software quality and maintainability, reducing the risk of software failure, and it is a primary requirement to guide the subsequent steps in the refactoring process. Detecting code smells is not only about fixing immediate issues but also about continuous improvement. By regularly monitoring and addressing code smells, developers can learn from past mistakes, refine their coding practices, and evolve as software engineers. This iterative process fosters a culture of quality and craftsmanship, leading to better code quality and more efficient development practices over time[28]. Detection rules of code smells are approaches used to detect code smells through a combination of different software metrics with predefined threshold values. Most approaches for code smell detection use object-oriented metrics to determine if a software system contains code smells or not[14]. Most current detectors need the specification of thresholds that allow them to distinguish smelly and non-smelly codes[37]. Many approaches have been presented by the authors for uncovering the smells from the software systems. Different detection methodologies differ from manual to visualization-based, semi-automatic studies, automatic studies, empirical-based evaluation, and metrics-based detection of smells. Most techniques used to detection of code smells rely on heuristics and discriminate code artifacts affected (or not) by a particular type of smells through the application of detection rules which compare the values of metrics extracted from source code against some empirically identified thresholds. Researchers recently adopted ML techniques to detect code smells to avoid thresholds and decrease the false positive rate in code smell detection tools [38], [39].

2.3 Software Metrics

Software Metrics play the most vital role in building a prediction model to improve software quality by predicting as many software defects as possible. Software metrics are essential aids in measuring and improving software quality, which are used to measure and characterize software engineering products[34]. Software metrics can be used to collect information regarding the structural properties of a software design, which can be further statistically analyzed, interpreted, and linked to its quality. Software metrics provide quantitative data that can be analyzed to identify potential areas of concern. By measuring various aspects of the codebase, such as complexity, size, or adherence to coding standards[40]. Software metrics help identify patterns and indicators associated with software bugs or code smells. By analyzing historical data and correlating software metrics with known issues, developers can spot recurring patterns or combinations of software metrics that indicate potential problems. This enables them to proactively address these areas to prevent software bugs or improve code quality. Moreover, software metrics support decision-making in bug prevention and code quality improvement efforts. By utilizing software metrics, developers can make informed decisions regarding code refactoring, architectural changes, or allocation of resources to

address code smells and potential bug-prone areas effectively [41], [42]. Software metrics can be classified as static code metrics and process metrics. Static code metrics can be directly extracted from source code, like Lines of Code (LOC), and Cyclomatic Complexity Number (CCN). Object-oriented metrics are a subcategory of static code metrics, like Depth of Inheritance Tree (DIT), Coupling Between Objects (CBO), Number of Children (NOC), and Response for Class (RFC)[4]. Object-oriented metrics are often used to assess testability, maintainability, or reusability of source code[18]. Tables 2.1 and 2.2 show the static code metrics. Process metrics can be extracted from the source code management system based on historical changes in source code over time. These metrics reflect the modifications over time, e.g., changes in source code, the number of code changes, developer information, etc.[43], [44]. Several researchers in the primary studies used McCabe and Halstead metrics as independent variables in the studies of software bug and code smells. The first use of McCabe metrics was to characterize code features related to software quality. McCabe's has considered four basic software metrics: cyclomatic complexity, essential complexity, design complexity, and lines of code[45]. Halstead also considered that the software metrics fall into three groups: base measures, derived measures, and line of code measures [46], [47]. Table 2.3 shows McCabe's and Halstead metrics. Metrics can also be classified based on the development phase of the software life cycle, into source code level metrics, detailed design level metrics, or test level metrics [48], [49].

Table 2.1 Show the static code metrics

Size	Complexity	Cohesion	Coupling	Encapsulation	Inheritance
Lines of Code (LOC)	McCabe's CYCLOmatic complexity (CYCLO)	Lack of Cohesion between Methods (LCOM)	Class Fan Out Complexity (CLASS_FAN_OUT)	Locality of Attribute Accesses (LAA)	Depth of Inheritance Tree (DIT)
Lines of Code Excluding Accessor and Mutator Methods (LOCNAMM*)	Weighted Methods per Class (WMC)	Tight Class Cohesion (TCC)	Access To Foreign Data (ATFD)	Number of Accessor Methods (NOAM)	Response for a Class (RFC)
Number of Methods (NOM)	Weighted Methods Count of Not Accessor or Mutator Methods (WMCNAMM*)		Foreign Data Providers (FDP)	Number of Public Attributes (NOPA)	Number of Children (NOC)
Number of Packages (NOPK)	Average Methods Weight of Not Accessor or Mutator Methods (AMWNAMM*)		Coupling Between Objects (CBO)		Number of Methods Overridden (NMO)
Number of Classes (NOCS)	Average Methods Weight (AMW)		Called Foreign Not Accessor or Mutator Methods (CFNAMM*)		Number of Inherited Methods (NIM)
Number of Methods Excluding Accessor and Mutator Methods (NOMNAMM*)	Maximum Nesting Level of Control Structures (MAXNESTING)		Coupling Intensity (CINT)		Number of Implemente d Interfaces (NOII)

Number of Attributes (NOA)	Weight of Class (WOC)		Coupling Dispersion (CDISP)		
	Called Local Not Accessor or Mutator Methods (CLNAMM)		Maximum Message Chain Length (MaMCL§)		
	Number of Parameters (NOP)		Mean Message Chain Length (MeMCL§)		
	Number of Accessd Variables (NOAV)		Number of Message Chain Statements (NMCS§)		
	Access To Local Data (ATLD*)		Control Coupling (CC)		
	Number of Local Variable (NOLV)		Number of Methods Affected by the Measured Method (CM)		

Metrics having a “*” in the name are customized versions of standard metrics, or slight modifications of original metrics. Metrics with a “§” suffix, refer to metrics that have been defined specifically for detecting the Message Chain code smell.

Table 2.2 Description list of 20 traditional static code metrics

Attribute	**Description**
dit	The maximum distance from a given class to the root of an inheritance tree
noc	Number of children of a given class in an inheritance tree
cbo	Number of classes that are coupled to a given class
rfc	Number of distinct methods invoked by code in a given class
lcom	Number of method pairs in a class that do not share access to any class attributes
lcom3	Another type of the lcom metric proposed by Henderson–Sellers
npm	Number of public methods in a given class
loc	Number of lines of code in a given class
dam	The ratio of the number of private/protected attributes to the total number of attributes in a given class
moa	Number of attributes in a given class that are of user-defined types
mfa	Number of methods inherited by a given class divided by the total number of methods that can be accessed by the member methods of the given class
cam	The ratio of the sum of the number of different parameter types of every method in a given class to the product of the number of methods in the given class and the number of different method parameter types in the whole class
ic	Number of parent classes that a given class is coupled to
cbm	Total number of new or overwritten methods that all inherited methods in a given class are coupled to
amc	The average size of methods in a given class
ca	Afferent coupling, which measures the number of classes that depend on a given class
ce	Efferent coupling, which measures the number of classes that a given class depends on
max_cc	The maximum McCabe's cyclomatic complexity (CC) score of methods in a given class
avg_cc	The arithmetic mean of McCabe's cyclomatic complexity (CC) scores of methods in a given class

Table 2.3 Descriptions of McCabe's and Halstead Metrics

Metrics	**Type**	**Description**
Loc	McCabe	It counts the line of code in software module.
v(g)	McCabe	Measure McCabe Cyclomatic Complexity.
ev (g)	McCabe	McCabe Essential Complexity.
iv (g)	McCabe	McCabe Design Complexity.
N	Derived Halstead	Total number of operators and operands.

V	Derived Halstead	Volume.
L	Derived Halstead	Program length.
D	Derived Halstead	Measure difficulty.
I	Derived Halstead	Measure Intelligence.
E	Derived Halstead	Measure Effort.
B	Derived Halstead	Effort estimate.
T	Derived Halstead	Time Estimator.
Locoed	Line Count	Number of lines in software module.
Locomment	Line Count	Number of comments.
Loblank	Line Count	Number of blank lines.
Locodeandcomment	Line Count	Number of codes and comments.
uniq_op	Basic Halstead	Unique operators.
uniq_opnd	Basic Halstead	Unique operands.
total_op	Basic Halstead	Total operators.
total_opnd	Basic Halstead	Total operands.
BranchCount	Branch	Total Number of branch count.

2.4 Summary

In this chapter, we have discussed the theoretical background and literature related to the fundamental concepts of our dissertation. We discussed the importance of the prediction of software bugs and code smells, the strategies and approaches used to predict software bugs and code smells, and software metrics used in the prediction of software bugs and code smells. While predicting software bugs and code smells have distinct focuses, they share a common goal of improving software quality. They both rely on indicators, adopt a proactive approach, use software metrics as indicators, and contribute to continuous improvement. By integrating the prediction of software bugs and code smells into the development process, developers can enhance software quality, prevent software bugs, and create more maintainable code. Overall, we realized that predicting software bugs and detecting code smells is crucial for cost-effective development, quality assurance, user satisfaction, security, reputation, and compliance. Developers can proactively identify and resolve software bugs to deliver higher-quality software that meets user expectations and industry standards. Additionally, software metrics play a crucial role in software development by providing quantitative data to support decision-making, track progress, and drive continuous improvement. So, predicting software bugs and detecting code smells based on software metrics are essential for developing high-quality, reliable, and maintainable software products.

Chapter 3 Artificial Intelligence (AI)

This chapter provides an overview of artificial intelligence techniques. It aims to equip readers with a fundamental understanding of the various approaches and methodologies that form the backbone of AI applications. Moreover, particular emphasis is placed on the artificial intelligence techniques utilized in this research, such as Machine Learning (ML) and Artificial Neural Networks (ANNs).

3.1 Artificial Intelligence Techniques

The field of Artificial intelligence (AI) is witnessing a recent upsurge in research, tools development, and deployment of applications[23]. AI is being widely adopted and incorporated into almost every kind of software application. where software engineers need to have a thorough grasp of what AI is and understand how to incorporate AI into the software development lifecycle[50]. AI is a branch of Computer Science that pursues creating computers or machines as intelligent as human beings. AI is accomplished by studying how the human brain thinks and how humans learn, decide, and work while trying to solve a problem. AI techniques such as ML, Neural Networks, fuzzy logic, etc. have been advocated by many researchers and developers as the way to improve many of the software development activities. AI techniques, specifically, ML techniques are commonly used for the prediction of software bugs and code smells compared to other techniques such as manual code inspection or rule-based approaches because they offer automation, scalability, and a data-driven approach[51]. ML models can handle large codebases, learn from historical data and leverage code metrics for data-driven analysis, capturing complex patterns and dependencies that may not be apparent through traditional methods, and adapt to new patterns, making them effective in identifying software bugs and code smells that may be difficult to detect manually. They provide objective and consistent analysis, enable early detection and prevention of issues, allowing developers to address issues before they become critical. They optimize resource allocation by prioritizing bug fixes based on severity or impact. Overall, ML techniques enhance the accuracy, efficiency, and overall quality of the prediction of software bugs and code smells processes, making them valuable tools for software development. There are several ML techniques commonly used in the prediction of software bugs and code smells[52].

3.1.1 Machine Learning (ML)

Machine learning (ML) is an area of research where computer programs can learn and get better at performing specific tasks by training on historical data or study of computer algorithms that provide systems the ability to automatically learn and improve from experience[10]. It is generally seen as a sub-field of AI. ML algorithms can be applied to analyze data from different perspectives to allow developers to obtain useful information [53], [54]. ML algorithms allow the systems to make decisions autonomously without any external support. Such decisions are made by finding valuable underlying patterns within complex data. High quantities of data are needed to develop ML model-based prediction [55], [56]. ML algorithms build models from training examples, which are then used to make predictions when faced with new examples[30]. ML techniques can be categorized into supervised, unsupervised, and reinforcement [35], [37]. ML algorithms have received extensive attention in the field of software engineering for a considerable period. Therefore, recently ML algorithms have been adopted to enhance research tasks in the prediction of software bugs and code smells[9].

3.1.1.1 Supervised learning

Supervised Learning is the ML task of inferring a function from labeled training data which consists of a set of training examples. Supervised learning is applied when the data is in the form of input variables and output target values[56]. In supervised learning, the training dataset has an output variable that needs to be predicted or classified. All algorithms learn some kind of patterns from the training dataset and apply them to the test dataset for prediction or classification[57]. It has two known supervised learning tasks (classification, and regression). Classification concerns building a predictive model for function with discrete range, while regression concerns continuous range model building. Supervised learning is fairly common in classification problems because the goal is often to get the computer to learn a classification system that we have created[58]. The most commonly supervised ML methods include concept learning, classification, rule learning, instance-based learning, Bayesian learning, linear regression, neural network, SVM, etc.[56]. The following subsections describe the supervised ML techniques used in our research work.

3.1.1.1.1 Decision Tree (DT)

Decision Tree (DT) is a popular supervised machine-learning method used for the purpose of regression and classification[4]. It refers to a hierarchal model or a tree with decision nodes that have more than one branch and leaf nodes that represent the decision. Each node in a DT represents a feature in an instance to be classified, and each branch represents the value thresholds the contained nodes can assume. Instances are categorized beginning at the root node and sorted based on their attribute values [21], [59]. There are different types of decision trees. The classic among them is the ID3 (Iterative Dichotomiser 3), birthing trees by recursively choosing the best feature to split the data. C4.5, its successor, added the ability to handle continuous attributes and pruning to trim excessive branches. CART (Classification and Regression Trees) is another heavyweight, excelling in both classification and regression tasks. Chi-Square is one of the oldest tree classification methods. It determines the statistical significance of the differences between sub-nodes and parent nodes. It is measured as the sum of squares of standardized differences between observed and expected frequencies of the target variable. Random Forests brings a dash of unpredictability to the mix, employing an ensemble of decision trees for robust performance. On the other hand, Gradient Boosted Trees take a sequential approach, refining the mistakes of previous trees to boost accuracy. ID3 is the most common type of decision tree. In the ID3 DT, all features are set as a root node. After that, the features are divided by finding the Entropy that measures the harmony in the data; the entropy values are between 0 and 1[35], [37]. Mathematically, Entropy for one attribute is represented as:

$$E(F) = \sum_{i=1}^{c} - p_i \log_2 p_i \quad (1)$$

Where *C* is the number of outputs, p_i is the probability of occurrences of each output from all outputs, and *F* is a feature with some data.

3.1.1.1.2 Random Forest (RF)

Random Forest (RF) is one of the most utilized models due to its effortlessness and the way it can be used for characterization and relapse assignments. It is adaptable and simple to utilize ML calculation, even without hyper-parameter tuning[35]. RF classifier is a special case of

Bagging consisting of a collection of tree-structured classifiers. RF selects random features to create bootstrap models using DT. RF algorithm considers K randomly chosen attributes at each node to construct a classification tree. In the classification setting, the prediction of the RF is the most dominant class among predictions by individual trees[60]. If there are T trees in the forest, then the number of votes received by a class m is:

$$U_m = \sum_{t=1}^{T} I(\check{C}_t == m) \tag{2}$$

where $\check{C}_t$ is the prediction of the *t* tree on a particular instance. The indicator function $I(\check{C}_t == m)$ takes on the value *1* if the condition is met, else it is zero.

3.1.1.1.3 Naïve Bayes (NB)

Naïve Bayes (NB) is a supervised learning algorithm and defines as a simple probabilistic classifier and efficient based on the Bayes' theorem with an independence assumption between the features, this means that the Naive Bayes classifier is based on estimating the probabilities of the unobserved node, based on the observed probabilities [21], [61]. Bayes' theorem finds the probability of an event occurring given the probability of another event that has already occurred. Bayes' theorem is stated mathematically as the following equation:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{3}$$

In the above equation, using Bayes' theorem, we can find the probability of *A*, given that *B* occurred. *A* is the hypothesis, and *B* is the evidence, $P(B|A)$ is the probability of *B* given that *A* is True, *P(A)* and *P(B)* are the independent probabilities of *A* and *B*.

3.1.1.1.4 Support Vector Machine (SVM)

Support Vector Machine (SVM) is one of the regulated ML models which is, for the most part, utilized for classification and relapse investigation. The primary role is to discover a hyper-plane, which divides the dimensional data completely into two categories [2], [62]. SVMs are based on a "margin" on either side of a hyperplane separating two features. Its optimizing objective is to increase the margin and create the most significant distance between features in the hyperplane. Complexity is not affected by the number of features. So SVM is appropriate for learning tasks where the number of features is so much concerning the number of training instances. The principal objective of SVM is to outline a model that predicts the dataset's target estimation in the testing stage. Subsequently, SVM becomes a decent contender for planning a model in anticipating issue-inclined modules[63]. The general type of SVM work is defined as:

$$F(x) = W * Q(x) + b \tag{4}$$

Where *w* is a weight vector, *x* is the input vector, *b* is the intercept and bias term of the hyperplane equations.

3.1.1.1.5 K-Nearest Neighbor (K-NN)

K-Nearest Neighbor (K-NN) define as a simple supervised classification algorithm in which an object is classified by looking at the K nearest objects and by choice of the most frequently occurring class[64]. It is also a lazy-learning technique that classifies elements based on their position and space in a hyperplane. Since in the K-NN algorithm, we need k nearest points.

Thus, the first step is calculating the distance between the input data point and other points in our training data[63]. The distance between these two points is:

$$d(x, y) = \sqrt{\sum_{i=1}^{p} (x_i - y_i)^2} \quad (5)$$

Suppose x is a point with coordinates $(x_1, x_2, ..., x_p)$ and y is a point with coordinates $(y_1, y_2, ..., y_p)$.

3.1.1.1.6 Logistic Regression (LR)

Logistic Regression (LR) is a popular statistical model used for binary classification problems, where the goal is to predict the probability of an instance belonging to a certain class. It models the relationship between the input features and the probability of the positive class using a logistic function. Logistic regression uses a logistic function called a sigmoid function to map predictions and their probabilities. The sigmoid function refers to an S-shaped curve that converts any real value to a range between 0 and 1 [61], [64]–[66]. The sigmoid function is referred to as an activation function for logistic regression and is defined as:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (6)$$

Where $f(x)$ is the predicted probability that the target variable y belongs to the positive class, given the feature value x, e is the base of the natural logarithm (approximately 2.71828). In many cases, multiple explanatory variables affect the value of the dependent variable. To model such input datasets, logistic regression formulas assume a linear relationship between the independent variables. The sigmoid function can be modified, and the final output variable calculated as:

$$y = f(\beta_0 + \beta_1 * x_1 + \beta_2 * x_2 + ... + \beta_n * x_n) \quad (7)$$

Where $\beta_0, \beta_1, \beta_2, ..., \beta_n$ are the coefficients (also known as weights or parameters) associated with each feature, $x_1, x_2, ..., x_n$ are the feature values.

3.1.1.1.7 XGBoost

XGBoost (XGB) is one of the recently introduced robust ML algorithms. XGB is a powerful gradient boosting algorithm that is widely used for supervised learning tasks such as regression and classification. It is known for its high predictive performance and efficient computation[63]. The formula for the XGB model is given as:

$$\tilde{y}_i = F(x_i) = b + n \sum_{k=1}^{K} f_k(x_i) \quad (8)$$

where b is the base prediction, n is the learning rate hyperparameter that helps control overfitting by reducing the contributions of each booster, and each of the K boosters f_k is a decision tree.

3.1.1.2 Unsupervised learning

Unsupervised Learning is also called learning from observation. Unsupervised learning is applied when the data is available only in the form of an input and there is no corresponding output variable. Such algorithms model the underlying patterns in the data in order to learn more about its characteristics[56]. Unsupervised learning seems much harder: the goal is to have the computer learn how to do something that we don't tell it how to do[58]. In

unsupervised learning, the system has to explore any patterns based only on the common properties of the example without knowing how many or even if there are any patterns. The most common methods in unsupervised learning are association rule mining, sequential pattern mining, and clustering[67].

3.1.1.3 Reinforcement learning

Reinforcement learning is somewhere between supervised and unsupervised learning[68]. Reinforcement learning is applied when the task at hand is to make a sequence of decisions toward a final reward[56]. Where the algorithm learns a policy of how to act given an observation of the world. Every action has some impact on the environment, and the environment provides feedback that guides the learning algorithm[58]. During the learning process, an artificial agent gets either rewards or penalties for the actions it performs. Its goal is to maximize the total reward. In reinforcement learning, the algorithm gets told when the answer is wrong but does not get told how to correct it. It has to explore and try out different possibilities until it works out how to get the answer right[68]. Examples include learning agents to play computer games or performing robotics tasks with end goals[56].

3.1.2 Artificial Neural Networks (ANNs)

Artificial neural networks (ANNs) are biologically inspired computer software built to imitate the way in which the human brain processes information[21]. ANNs are ML models or nonlinear classifiers used to model complex relationships between inputs and outputs. An ANNs model contains multiple units (layers) for information processing which are known as neurons. The layers are typically named the input layer, hidden layer, and output layer [69], [70]. The typical architecture of ANN is shown in Figure 3.1. When implementing a neural network, a set of consistent training values must be available to set up the expected operation of the network and a set of validation values to validate the training process[71]. ANNs collect knowledge by detecting the patterns and relationships in data and learning or training through experience. When neural networks are used for data analysis, it must be important to distinguish between ANN Models which refer to the network's arrangement, and ANN Algorithms which refer to computations that eventually produce the network outputs. There are two approaches to training ANNs: supervised and unsupervised. The most often used ANNs for prediction and classification tasks is a fully connected and supervised network with a backpropagation learning rule. During the learning stage, the weights of each neuron are considered and adjusted according to the requirements. To obtain the final weight for neurons, each neuron gives input to each preceding layer, and later these inputs are multiplied by their weight. According to this process, the neuron computes the activation level from this sum, and the output is sent to the following layer where the final solution is estimated [28], [34]. The output of a neuron that is in the layer can be described by the equation below:

$$\mathrm{Y_i} = \mathrm{f_i} \left(\sum_{\mathrm{j=1}}^{\mathrm{n}} \mathrm{w_{ij}} * \mathrm{x_j} + \mathrm{b_i} \right) \tag{9}$$

where Y_i represents network output, n is the total number of inputs to this neuron, x_j represents network input, w_{ij} is the connection weights between input and output nodes, b_i is the bias and f_i is the transfer function.

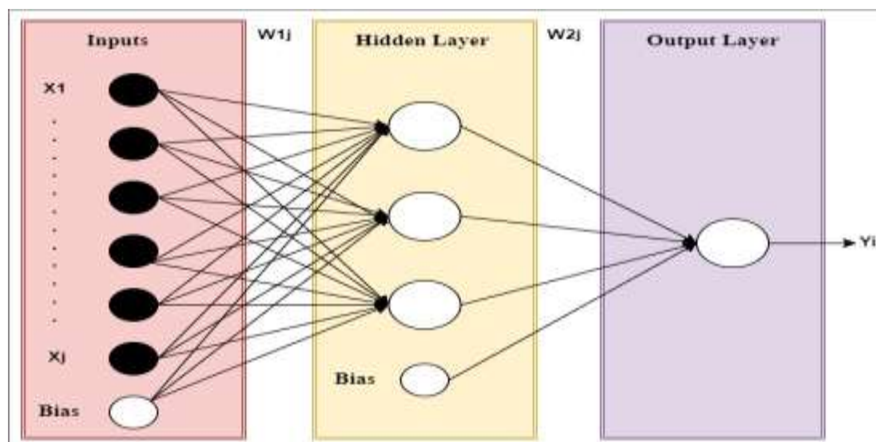


Figure 3.1 The typical ANN architecture[70]

3.1.2.1 Multi-layer Perceptron (MLP)

A Multi-layer Perceptron (MLP) network is a particular type of artificial neural network that consists of different layers (input layer, hidden layer, and output layer). It was created to solve nonlinear classification problems that cannot be solved by a single layer. A multilayer neural network consists of many units (neurons) joined together in a pattern of connections[37]. It uses nodes with a specified weight to connect the layers. Each node is a neuron that utilizes a nonlinear activation function. The backpropagation algorithm is used to train the model in the multilayer perceptron network[5]. The formula of the multilayer perceptron network model is as follows:

$$f(x) = \left(\sum_{i=1}^{m} w_i * x_i \right) + b \quad (10)$$

Where m is the number of neurons in the previous layer, w_i is a random weight, x_i is the input value, b is a random bias.

3.1.2.2 Deep learning (DL)

Deep learning (DL) algorithms have received extensive attention in the field of software engineering for a considerable period. DL is one of the AI functions that mimic the workings of the human brain. It allows and helps to solve complex problems by using a data set that is very diverse, unstructured, and interconnected[72]. DL is a type of ML that allows computational models consisting of multiple processing layers to learn data representations with multiple levels of abstraction. DL architecture has been widely used to solve many detections, classification, and prediction problems[73]. There are many activation functions used in DL such as sigmoid, Rectified Linear unit (Relu), and Hyperbolic Tangent (Tanh). Activation functions are a critical component of DL, serving as the nonlinearities that allow neural networks to model complex relationships in data. Their importance lies in their ability to introduce non-linearity, control gradient flow during training, and adapt the network's behaviour to different problem domains. The right choice of activation function can significantly impact training speed, model performance, and the ability to capture intricate patterns in data. Whether it is the efficiency of ReLU, the sigmoid's interpretability, or the tanh's versatility, selecting the appropriate activation function is a key decision in designing neural networks. Therefore, activation functions enable the training of the DL model quickly and accurately. Relu and sigmoid [74], [75] are the most common activation functions used in

DL. So, in our proposed models, we used the Relu activation function for the inputs and hidden layers and the Sigmoid activation function for the output layer. The equations to calculate Relu and sigmoid are as follows:

$$h_i^{m} = \mathrm{ReLU}(W_i^{m-1} \times V_i^{m-1} + b^{m-1}) \tag{11}$$

where h_i^{m} represents convolutional layer, W_i^{m-1} represents the weights of neuron, V_i^{m-1} represents the nodes, and b^{m-1} represents the bias layer.

$$S(x) = \frac{1}{1 + e^{-\sum_k W_i + X_i + b}} \tag{12}$$

where X_i represents the input, W_i is the weight of the input and b is the bias.

3.1.2.2.1 Convolutional Neural Network (CNN)

Convolutional Neural Network (CNN) is a special type of deep neural network, or a class of convolutional feedforward neural networks used to process data that has a known, grid-like topology. It is constructed to mimic the visual perception of biological processes and can be used for both supervised learning and unsupervised learning[76]. CNN has been tremendously successful in practical applications, including speech recognition, image classification, and natural language processing [77], [78]. The CNN model is inspired by the typical CNN architecture used in image classification and consists of a feature extraction part and a classification part, as shown in Figure 3.2. These parts consist of multiple layers of convolution, batch normalization, and maximum merge layers. These layers constitute the hidden layer of the architecture. Convolution is a fundamental operation enabling the network to detect and learn relevant features within the input data automatically. Convolutional layers employ small learnable filters or kernels to slide over the input. Each filter is a small matrix (usually 3x3 or 5x5) that slides over the input data. These filters capture specific features such as edges, textures, or more complex patterns. This process of convolution generates feature maps that highlight where these patterns are found in the input, while the maximum pooling layer achieves a reduction in the dimension of the feature space. Batch normalization is used to mitigate the effect of different input distributions for each training mini batch for the purpose of improving training [79], [80].

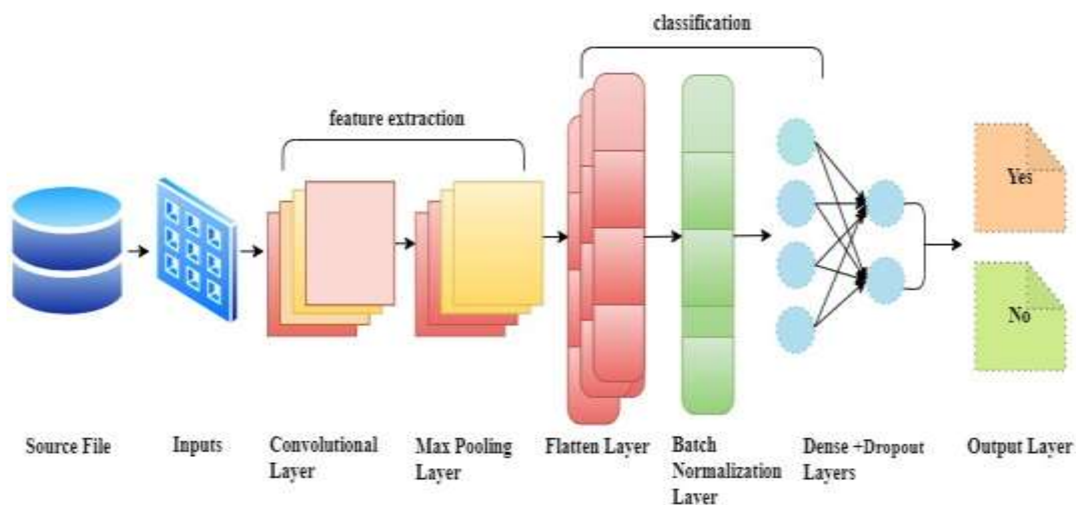


Figure 3.2 The typical CNN architecture[80]

3.1.2.3 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a type of ANNs that can process a sequence of inputs and retain its state while processing the next sequence of inputs and can efficiently acquire the nonlinear features that are in order. Where the nodes and their connections form a temporally directed graph along a temporal sequence [81], [82]. RNN is widely used to solve many different problems, such as pattern recognition, identification, classification, vision, speech, control systems, etc.[83]. Due to the problem of long-term dependencies that arise when the input sequence is too long, RNN cannot guarantee a long-term nonlinear relationship. This means that there is a gradient vanishing and gradient explosion phenomenon in the learning sequence. RNNs can use memory units (internal state) to learn the relationship between the sequence pieces, making it possible for RNNs to capture contextual features of the sequence[84]. Many optimization theories and improved algorithms have been introduced to solve this problem such as Long-Short-Term-Memory (LSTM) networks, Bidirectional LSTM, Gated Recurrent Unit (GRU) networks, echo state networks, Independent RNN, etc. Standard RNNs take sequences as inputs, and each step of the sequence refers to a certain moment[85]. For a certain moment t, the output h_t not only depends on the current input x_t but is also influenced by the output from the previous moment $t-1$. The output of moment (t) can be formulated as the following equation:

$$h_t = f(U \times x_t + W \times h_{t-1} + b) \tag{13}$$

Where *U* and *W* denote the weights of the RNN, *b* denotes the bias, *f* is the activation function of the neurons.

3.1.2.3.1 Long-Short-Term-Memory (LSTM)

Long-Short-Term-Memory (LSTM) networks are a special type of RNN designed to recognize patterns in data sequences. LSTM networks were introduced to avoid or handle long-term dependency problems without being affected by an unstable gradient[55]. This problem frequently occurs in regular RNNs when connecting previous information to new information[48]. LSTM networks offer a set of key features that distinguish them in the realm of RNNs. Their primary strengths lie in their ability to capture long-term dependencies in sequential data, thanks to memory cells and gating mechanisms that control information flow. LSTMs incorporate three essential gates: the forget gate, which decides what to discard from the previous state; the input gate, responsible for selectively updating the memory cell with new information; and the output gate, which regulates the information output as the hidden state. Due to the ability of the LSTM network to recognize longer sequences of time-series data, LSTM models can provide high predictive performance[84]. Figure 3.3 shows the interacting layers of the repeating module in LSTM Networks. The cell state carries the information from the previous moments and will flow through the entire LSTM chain, which is the key that LSTM can have long-should be filtered from the previous moment, the output of the forget gate can be formulated as the following equation:

$$f_t = \sigma(W_f \,.\, [h_{t-1}, x_t] + b_f) \tag{14}$$

Where σ denotes the activation function, W_f and b_f denote the weights and bias of the forget gate, respectively. The input gate determines what information should be kept from the current moment, and its output can be formulated as the following equation:

$$i_t = \sigma(W_i . [h_{t-1}, x_t] + b_i) \tag{15}$$

Where σ denotes the activation function, W_i and b_i denote the weights and bias of the input gate, respectively. With the information from forget gate and input gate, the cell state C_{t-1} is updated through the following formula:

$$\check{C}_t = \tanh(W_c . [h_{t-1}, x_t] + b_c) \tag{16}$$
$$\check{C}_t = f_t \times C_{t-1} + i \times \check{C}_t)$$

$\check{C}_t$ is a candidate value that is going to be added into the cell state and C_t is the current updated cell state. Finally, the output gate decides what information should be output according to the previous output and current cell state.

$$o_t = \sigma(W_o . [h_{t-1}, x_t + b_o] \tag{17}$$
$$h_t = o_t \times \tanh(C_t)$$

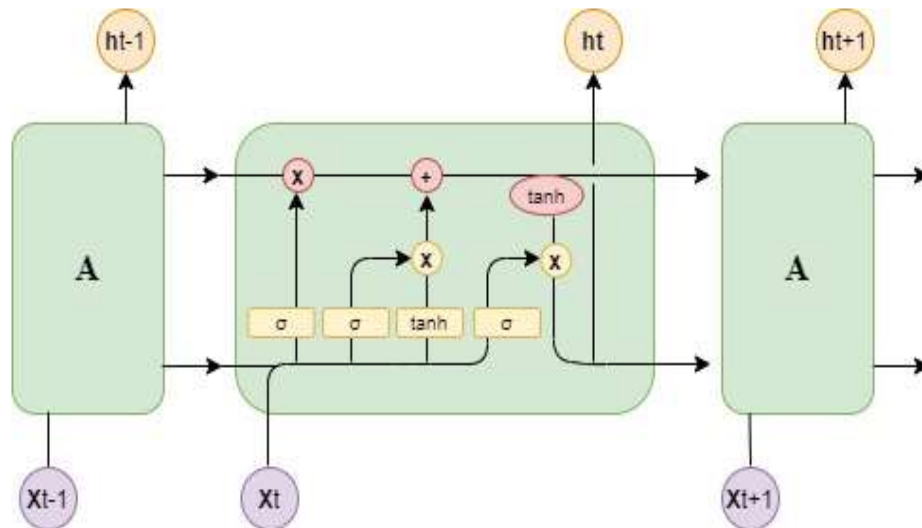


Figure 3.3 Interacting layers of the repeating module in an LSTM Networks[40]

3.1.2.3.2 Bidirectional Long-Short-Term-Memory (Bi-LSTM)

The idea behind Bidirectional Long-Short-Term-Memory (Bi-LSTM) networks is to exploit spatial features to capture bidirectional temporal dependencies from historical data to overcome the limitations of traditional RNNs [73], [86], [87]. Bi-LSTM networks are a new way to train data by expanding the capabilities of LSTM networks[84]; it uses two separate hidden layers to train the input data twice in the forward and backward directions, as shown in Figure 3.4. With the regular LSTM Networks, the input flows in one direction, either backward or forward. Bi-LSTM Networks are the process of making any neural networks have the sequence information in both directions (a sequence processing model that consists of two LSTM): one taking the input in a forward direction (past to future), and the other in a backward direction (future to past) [2].

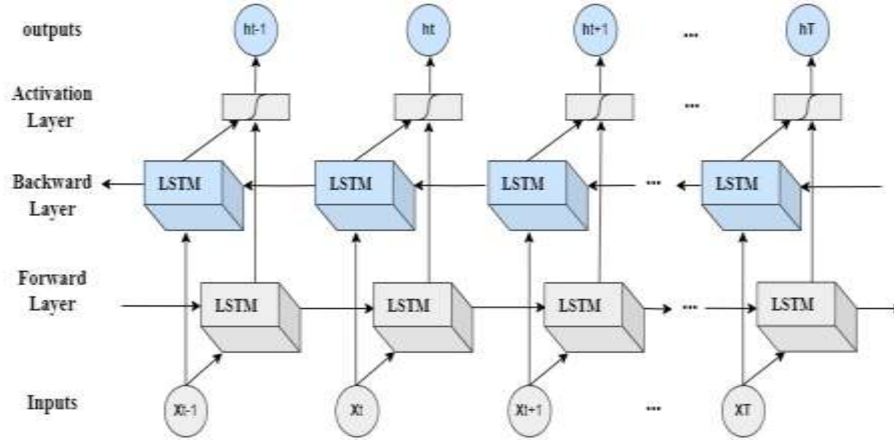


Figure 3.4 Interacting layers of the repeating module in a Bi-LSTM Network[86]

3.1.2.3.3 Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU) network is one of the optimized structures of the RNN[73]. The goal of the GRU network is to solve the long-term dependence and gradient disappearance problem of RNN[7]. The GRU is like LSTM in a forget gate but has fewer parameters than LSTM and uses an update gate and reset gate as shown in Figure 3.5. The GRU network uses the update and reset gates to improve and optimize the learning mechanism[83]. The update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future and the reset gate helps the model to decide how much of the past information to forget. Due to the ability of the GRU network to recognize longer sequences of time-series data, it can provide high predictive performance [84], [88], [89]. The update gate model in the GRU network is calculated as shown in the equation below.

$$z(t) = \sigma(W(z).[h(t-1), x(t)] + b_z) \tag{18}$$

the $z(t)$represents the update gate, $h(t-1)$ represents the output of the previous neuron, $x(t)$represents the input of the current neuron, $W(z)$represents the weight of the update gate, b_z is the bias for the update gate, and σ represents the sigmoid function. The reset gate model in the GRU neural networks is calculated as shown in equation below.

$$r(t) = \sigma(W(r).[h(t-1), x(t)] + b_r) \tag{19}$$

$r(t)$represents the reset gate, $h(t-1)$ represents the output of the previous neuron, $x(t)$represents the input of the current neuron, $W(r)$represents the weight of the reset gate, b_r is the bias for the reset gate, and σ represents the sigmoid function. The output value of the GRU hidden layer is shown in equation below.

$$\check{h}(t) = \tanh\left(W\check{h}.[rt * h(t-1), x(t)]\right) \tag{20}$$

$\check{h}(t)$represents the output value to be determined in this neuron, $h(t-1)$represents the output of the previous neuron, $x(t)$represents the input of the current neuron, $W\check{h}$ represents the weight of the update gate, and *tanh ()* represents the hyperbolic tangent function. rt is used to control how much memory needs to be retained. the hidden layer information of the last output as shown in equation below.

$$h(t) = \left(1 - z(t)\right) * h(t-1) + z(t) * \check{h}(t) \tag{21}$$

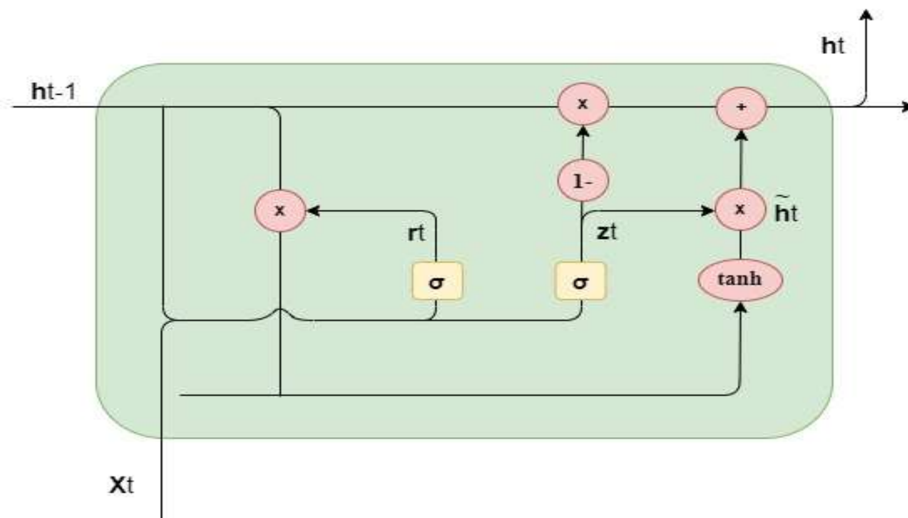


Figure 3.5 Interacting layers of the repeating module in a GRU Networks[7]

3.2 Summary

In this chapter we have provided an overview of artificial intelligence techniques, in particular, ML techniques. Specifically, we focused on describing ML techniques that are commonly used in the literature for the prediction of software bugs and code smells. We concluded that ML techniques have recently gained attention in the literature for the prediction of software bugs and code smells due to their ability to recognize patterns, automate processes, handle large-scale data, adapt to different contexts, continuously improve, and complement static analysis. ML models can analyze code metrics, historical bug data, or code smells indicators to identify patterns that indicate the presence of software bugs or code quality issues. By automating the analysis, ML techniques save time and effort for developers. ML models are scalable, adaptable to different coding styles and programming languages, and can continuously learn and improve over time. They complement static analysis tools by providing a more comprehensive analysis of code quality. While ML techniques are not infallible and require domain expertise for interpretation, they offer valuable insights and support in creating more reliable and maintainable software.

Chapter 4 Data Imbalance and Data-Balancing Methods

This chapter offers a concise introduction to the concept of data imbalance and data-balancing methods, with a special emphasis on data sampling methods.

4.1 Data Imbalance

The data imbalance problem is a hot topic being investigated recently by ML and data mining researchers, especially in the context of the prediction of software bugs and code smells. It is considered one of the current research topics of interest in supervised classification that frequently appears in several real-world datasets[90]. The main characteristic of the imbalanced data is class imbalances. The class imbalance can be intrinsic property or due to limitations to obtaining data such as cost, privacy, and large effort[13]. The class imbalance problem occurs when, in a dataset, one of the classes has fewer instances, usually called the minority class, than the other class, usually called the majority class[91]. In bug prediction, this means that the dataset may have a significantly higher number of non-buggy instances compared to buggy instances, while in code smells, certain types of code smells may be underrepresented compared to others[92]. This problem produces a poor classification rate for the minority class, which is usually the most important. Consequently, it becomes difficult for a classifier to effectively discriminate between the minority and majority classes, especially if the class imbalance is extreme, which has aroused the interest of many researchers to solve the problem of class imbalance[93].

4.2 Data-Balancing Methods

Data imbalance is a common challenge in the prediction of software bugs and code smells tasks, where certain classes of interest are underrepresented compared to others. Data-balancing methods are crucial in addressing this issue and improving the performance and accuracy of the models[13]. By balancing the data, these methods help in achieving improved model performance, avoiding bias in predictions, enhancing the detection of rare events, preventing overfitting, and providing valuable insights into software bugs and code smells. Overall, data-balancing ensures that the models are trained on a more representative distribution of instances, leading to more accurate and reliable predictions in the prediction of software bugs and code smells tasks. Several data-balancing techniques have been developed to overcome the class imbalance problem, these techniques include subset methods, cost-sensitive learning, algorithm-level implementations, ensemble learning, feature selection methods, sampling methods, etc.[15]. These techniques can be grouped into two distinct categories: external methods that use existing algorithms without modification (corresponds to methods that operate on the dataset in a preprocessing step preceding classification), and internal methods that create new algorithms or modify existing algorithms to take into account class imbalances (modifies the classification algorithm in order to put more emphasis on the minority class), the two types of methods can be roughly divided into data level and algorithm level [91], [93]. The most common techniques used in previous work to deal with the class imbalance problem are external methods which are based on the data sampling technique (Oversampling and Undersampling methods) [87], [94].

4.2.1 Data Sampling (Resampling) Methods

Data sampling techniques are more prevalent in the studies of the prediction of software bugs and code smell due to their easy employment and independence (i.e., they can be applied to any prediction model)[87]. Therefore, data sampling techniques are commonly used to address the class imbalance problem in ML. These techniques are popular due to their simplicity, compatibility with various algorithms, computational efficiency, and retention of information. Data sampling methods are relatively easy to understand and implement, work well with different learning algorithms, and have minimal computational overhead[93]. Additionally, models trained on balanced data can provide more interpretable results. Data sampling methods tend to adjust the prior distribution of the majority and minority classes in the training data by either reducing the majority class instances or increasing the minority class instances to obtain a balanced class distribution and reduce the discrepancy among the sizes of the classes. There are three main categories of data sampling techniques that are: Oversampling Methods, Undersampling Methods, and Hybrid (Combined-Sampling Methods)[95]. Figure 4.1 shows how data sampling methods deal with class imbalance.

4.2.1.1 Undersampling Methods

Undersampling is a non-heuristic method where a subset of the majority class is chosen to create a balanced class distribution. The advantage of this method is that the elimination of some examples could significantly reduce the size of the data and therefore decrease the run-time cost, especially in the case of big data[95]. There are many Undersampling methods such as Random Undersampling, Near Miss, Tomek links, etc.

- Random Undersampling is an Undersampling method aiming to randomly eliminate samples of the majority class to obtain a balanced dataset[15]. This algorithm randomly removes samples of the majority class using either sampling with or without replacement[94], despite its simplicity, Random Undersampling is one of the most effective resampling methods [13], [15].
- Near Miss is an Undersampling method, which aims to balance class distribution by selecting examples based on the distance of majority class examples to minority class examples[96].
- Tomek links is a method of Undersampling developed by Tomek (1976) This algorithm works by deleting negative classes and positive classes further that have similar characteristics [95].

4.2.1.2 Oversampling Methods

Oversampling is a non-heuristic method used to address data imbalance in ML by increasing the number of instances in the minority class[15]. These methods aim to provide the model with more examples of the minority class, making it easier for the model to learn its patterns and improve its ability to classify it accurately [95], [97]. Oversampling methods are more effective than Undersampling methods in prediction accuracy[13]. There are many Oversampling methods such as Random Oversampling, Synthetic Minority Oversampling Technique (SMOTE), etc.

- Random Oversampling is a simple approach where we take samples at random from the small class and duplicate these instances so that it reaches a size comparable with the

majority class, it is defined as a method developed to increase the size of a training data set by making multiple copies of some minority classes[93].

- SMOTE is an Oversampling method based on creating synthetic instances for the minority classes. It is a method in which new samples of minority class are synthesized based on the feature space similarities among existing minority examples[87]. It is the most widely used and referenced method among the Oversampling methods[92]. The algorithm takes each minority class sample and introduces synthetic samples along the line joining the current instance and some of its k nearest neighbors from the same class. Depending on how much Oversampling is needed, the algorithm chooses randomly from the k nearest neighbors of them and forms pairs of vectors that are used to create the synthetic samples. The new instances create larger and denser decision regions. This helps classifiers learn more from the minority classes in those decision regions, rather than from the large classes surrounding those regions[93].

4.2.1.3 Hybrid (Combined-Sampling Methods)

Combined-sampling methods refer to the integration of multiple sampling techniques into a single approach (such as Oversampling and Undersampling) to improve the effectiveness and efficiency of the sampling process[98]. These methods aim to leverage the strengths of different sampling techniques while mitigating their limitations. There are various hybrid sampling methods, for example SMOTE Tomek method[95].

- SMOTE Tomek is a new technique that was applied using the library from imbalanced learn, which combines the SMOTE function for Oversampling and the Tomek Link function for Undersampling[99].

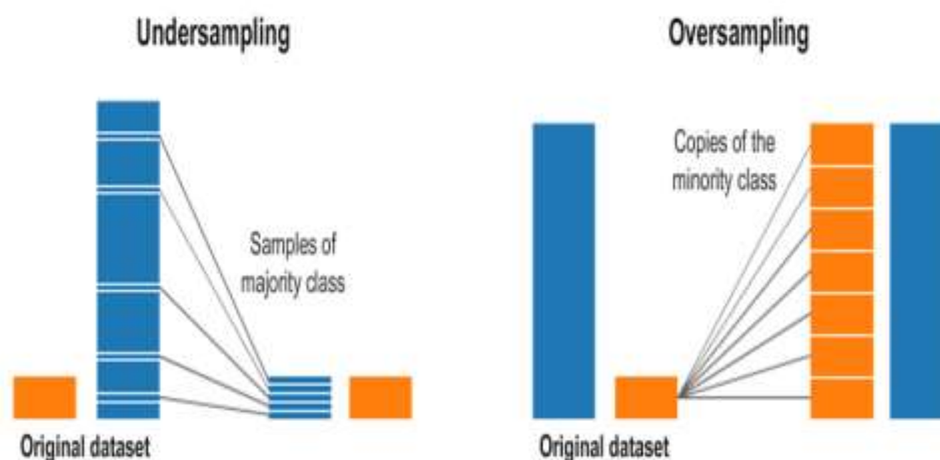


Figure 4.1 Shows how data sampling methods deal with class imbalance

4.3 Summary

In this chapter we have provided a short background of data imbalance and data-balancing methods. Specifically, we focused on describing data-balancing methods that are commonly used in the literature to address the problem of data imbalance in datasets of software bug and code smells. We concluded that data imbalance can pose challenges for ML models because they tend to favor the majority class and may struggle to adequately learn from the minority class. This can result in biased or inaccurate predictions, where the model may have high accuracy overall but performs poorly on the minority class or rare occurrences. In the context

of software bugs and code smells, this means that the model may have difficulties accurately identifying and predicting the occurrences of software bugs or specific code smells. Therefore, data imbalance should be addressed to ensure that the ML model can effectively learn from and make accurate predictions on all classes of interest, including the minority class instances. By applying data-balancing methods along with ML techniques in the prediction of software bugs and code smells, developers and analysts can build models that are more accurate, reliable, and unbiased. These methods help overcome the limitations of imbalanced datasets and ensure that the model's predictions are representative of the actual occurrence of software bugs and code smells in the software codebase.

Chapter 5 Proposed Methodology and Implementation

This chapter presents our proposed methodology and implementation, which describes the experiments performed. Several experiments and comparisons are conducted to predict software bugs and code smells based on ML techniques and data-balancing methods. The architecture of the methodology followed in the dissertation can be visualized in Figure 5.1.

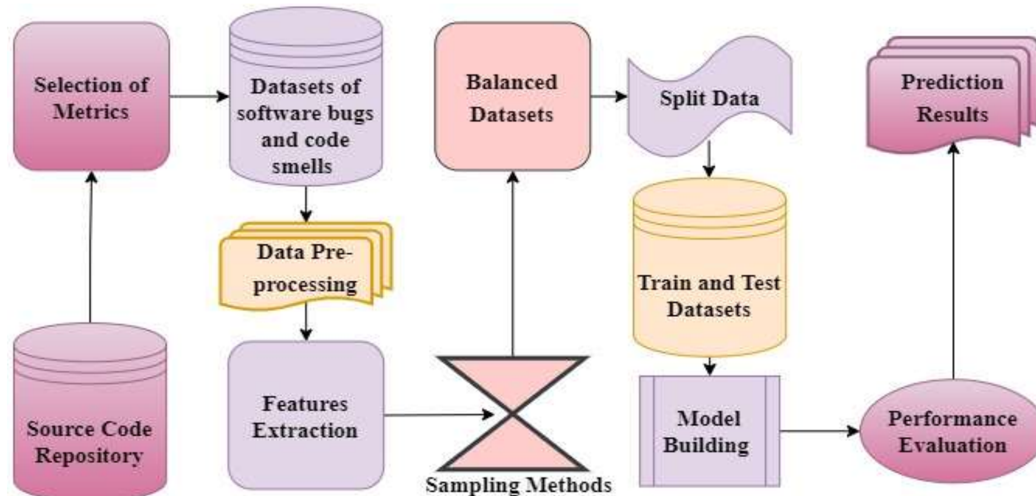


Figure 5.1 The architecture of the methodology followed in the dissertation

5.1 Experimental Design

This subsection presents the process of experimental design for our proposed approaches. We also discuss experimental design phases that are used in the experiments, such as proposed ML models, the data sets that are used to train and test the models, data pre-processing and features selection, data-balancing methods that are used to balance data sets, and performance measures that are used to evaluate and compare our proposed approaches with other existing approaches.

5.1.1 Proposed Approaches

In relation to software bug prediction, we developed four approaches. The first approach was developed based on four ML models which are DT, NB, RF, and LR. The second approach was developed based on combining two RNN models, namely LSTM and GRU, with an Undersampling method (Near Miss). The third approach was developed by combining a Bi-LSTM network with Oversampling methods (Random Oversampling and SMOTE). The fourth approach was developed using a combination method based on CNN and GRU with a hybrid sampling method (SMOTE Tomek).
Concerning code smell detection, we developed three approaches. The first approach was developed based on several ML algorithms which are DT, K-NN, SVM, XGB, and MLP combined with an Oversampling method (Random Oversampling). The second approach was developed based on a CNN combined with Oversampling method (SMOTE). The third approach was developed based on two RNN models (Bi-LSTM and GRU) combined with two sampling methods (Random Oversampling and Tomek links).

5.1.2 The Public Benchmark Datasets Used in This Research

When researching software bug prediction and code smell detection or related topics, it is essential to utilize appropriate data sets specifically designed for this purpose. To perform the experiments of this research and verify the validity of the proposed methods, the used datasets were obtained from the public benchmark datasets of software bugs and code smells that contain information for several projects. We used a public dataset because this is a benchmarking procedure for research on software bugs and code smells.

5.1.2.1 Software Bug Data Sets

We used three different public datasets to perform software bug prediction experiments. The first group was obtained from the NASA datasets, we selected four NASA public datasets, these datasets were collected from real software projects by NASA [100], [101]. Table 5.1 shows information about the NASA datasets. The second group was obtained from a public unified bug dataset, the authors considered 5 public datasets and downloaded the corresponding source code for each system in the datasets and source code analysis was performed to obtain a standard set of source code metrics. They have produced a unified bug dataset at the class and file level that is suitable for the building of new bug prediction models. Furthermore, they have compared the metric definitions and values of the different bug datasets[102]. The defective instances for the unified bug dataset (Class level metrics and File level metrics) are 8780 and 10240. While the non-defective instances are 38838 and 33504, respectively. Table 5.2 shows information about the public unified bug dataset. The third group was obtained from the PROMISE repository datasets. We selected six open-source Java projects from the PROMISE dataset. The source code and corresponding PROMISE data for all projects are public [47], [103], [104]. These projects cover applications such as XML parsers, text search engine libraries, and data transport adapters, and these projects have traditional static metrics for each Java file. To guarantee the generality of the evaluation results, experimental datasets consist of projects with different sizes and defect rates (in the six projects, the maximum number of instances is 965, and the minimum number of instances is 205. In addition, the minimum defect rate is 2.23% and the maximum defect rate is 92.19%). The defective instances for the PROMISE datasets (ant, camel, ivy, jedit, log4j, and xerces) are (166, 188, 40, 11, 16, and 151), respectively. While the non-defective instances are (579, 777, 312, 481, 189, and 437), respectively. Table 5.3 shows the essential information of selected projects, including project name, project version, number of instances, and defect rate or the percentage of defective instances.

Table 5.1 Description of the NASA datasets

Project Name	# Modules	% Defects	Language	Description
JM1	10885	19%	C	Real-time predictive ground system: Uses simulations to generate predictions.
PC1	1107	6.8%	C	Flight software for earth orbiting satellite.
KC1	2107	15.4%	C++	Storage management for receiving and processing ground data.
KC2	523	20%	C++	Software for science data processing.

Table 5.2 Description of the public unified bug dataset

Dataset	Software	Lines of code
PROMISE	Ant, Camel, Ckjm, Forrest, Ivy, JEdit, Log4J, Lucene, PBeans, Poi, Synapse, Velocity, Xalan, Xerces	2,805,253
Eclipse Bug Dataset	Eclipse	3,087,826
Bug Prediction Dataset	Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, Lucene, Mylyn	1,171,220
Bug catchers Bug Dataset	Apache Commons, ArgoUML, Eclipse JDT Core	1,833,876
GitHub Bug Dataset	Android Universal Image Loader, Antlr 4, Broadleaf Commerce, Ceylon IDE Eclipse Plugin, Elasticsearch, Hazelcast, JUnit, MapDB, mcMMO, MCT, Neo4J, Netty, OrientDB, Oryx, Titan	1,707,446

Table 5.3 Description of the PROMISE datasets

Project Name	Project Version	# Of Instances	Defect Rate %
ant	1.7	745	22.28%
camel	1.6	965	19.48%
ivy	2.0	352	11.36%
jedit	4.3	492	2.23%
log4j	1.2	205	92.19%
xerces	1.4	588	74.31%

5.1.2.2 Code Smells Data Sets

We used the proposed datasets in Arcelli Fontana et al [4] to perform code smell detection experiments. The authors selected 74 open-source systems from Qualitas Corpus as shown in Table 5.4. The Qualitas Corpus (QC) systems were collected by Tempero et al[105]. The QC systems comprise 111 systems written in Java belonging to different application domains and characterized by different sizes. The QC systems datasets consisted of 561 smelly instances and 1119 non-smelly instances. The first two datasets pertain to code smells at the class level, specifically for the god class (with 140 smelly cases and 280 non-smelly instances) and data class (with 140 smelly cases and 280 non-smelly instances). In contrast, the remaining two datasets focus on code smells at the method level: feature envy (with 140 smelly instances and 280 non-smelly instances) and long method (with 141 smelly instances and 279 non-smelly instances). The reason for selecting these datasets is that (i) the QC systems are the largest curated corpus for code analysis studies, with the current version having 495 code sets, representing 100 unique systems. The corpus has been successful in that groups outside its original creators are now using it, and the number and size of code analysis studies have significantly increased since it became available. (ii) Systems must be able to calculate metric values correctly. Moreover, these data sets are freely available, and researchers can iterate, compare and evaluate their studies. The selected metrics in QC systems are at class and method levels; the set of metrics is standard metrics covering different aspects of the code, i.e., complexity, cohesion, size, and coupling [4].

Table 5.4 Description of the Qualitas Corpus Systems

Number of systems	Lines of code	Number of packages	Number of classes
74	6,785,568	3420	51,826

5.1.3 Data Pre-processing

Pre-processing the collected data is one of the essential stages before constructing the model. To generate a good model, data quality needs to be considered. Not all data collected is suitable

for training and model building. Anyhow, the inputs will significantly impact the model's performance and later affect the output[106]. Data pre-processing is a group of techniques that are applied to the data to improve the data quality before model building to remove noise and unwanted outliers from the data set, dealing with missing values, feature type conversion, etc. Outliers are data points that deviate significantly from most of the data in a dataset. Detecting and handling outliers is crucial in data analysis and modelling, as they can disproportionately influence statistical measures and ML algorithms. Outliers can be detected using various methods, such as visual inspection of the data, statistical measures such as the Z-score or the interquartile range, or ML techniques. Once outliers are detected, they can be handled in various ways, such as removing them from the dataset, replacing them with the mean or median of the data, using outlier detection techniques using ML, or using algorithms less sensitive to outliers. All outliers in the data sets were treated by replacing them with the mean. All datasets are pre-processed by dealing with missing content and constant values. Handling missing values treatment improves performance measures and avoids biased results. Incomplete data can bias the results of the ML models and/or reduce the model's accuracy. Datasets used contain instances from different projects. Considering that, there are three main methods for handling missing data: deletion, imputation, and modelling. Deletion methods involve removing the missing values or the cases with missing values from the data set. Imputation means replacing the missing values with estimated values based on the available data. Modelling methods require incorporating the missing data mechanism into the analysis model or using methods that directly handle missing data. Missing values for the datasets used in this research are handled based on imputation methods, which means replacing them with the mean. In addition, instances are scaled to reduce the distance between independent variables. Normalization is necessary to convert the values into scaled values (transforming the features to be on a similar scale) to increase the model's efficiency. Therefore, the data set was normalized using Min–Max and Standard scaling. The formula for Min-Max scaling is given by (22), and the formula for Standard scaling is given by (23). After that, constant, quasi-constant and duplicated features are removed. It is followed by feature selection extracting feature subset that contributes maximum to the ML algorithms prediction variable[107].

$$X_{new} = (X - X_{min}) / (X_{max} - X_{min}) \quad (22)$$

Where *X*: It is a set of the observed values present in *X, X min*: It is the minimum values in *X* and *X max*: It is the maximum values in *X*.

$$X_{scaled} = X - \mu / \sigma \quad (23)$$

Where X_{scaled}: It is the scaled value*, X*: It is the original value, μ: It is the mean of the feature and σ: It is the standard deviation of the feature.

5.1.4 Features Selection

Feature selection is a critical process in ML that involves choosing the most relevant and informative features from the original set [108]. The objective is to enhance model performance, mitigate overfitting, and improve interpretability. Feature extraction facilitates the conversion of pre-processed data into a form that the classification engine can use [109], [110]. Feature selection in ML encompasses various methods, such as Filter Methods, Wrapper Methods, Embedded Methods, Dimensionality Reduction Techniques and Hybrid Methods aimed at identifying and utilizing the most relevant features for model training [111]. Filter

methods employ diverse criteria such as statistical tests, correlation coefficients, or information gain to rank and filter features based on their intrinsic characteristics, irrespective of the specific ML model. By efficiently screening out less informative or redundant features early in the process, filter methods help mitigate the curse of dimensionality and enhance computational efficiency. Wrapper methods in feature selection are dynamic techniques that assess the relevance of subsets of features by integrating them into the model training and evaluation process. Unlike filter methods that evaluate features independently, wrapper methods employ a trial-and-error approach, testing different combinations of features to identify the most informative subset. Standard wrapper methods include forward selection, backward elimination, and recursive feature elimination. Forward selection starts with an empty set and iteratively adds features based on their impact on model performance. In contrast, backward elimination begins with all features and progressively removes the least relevant ones. Recursive Feature Elimination recursively fits the model and eliminates the least significant feature in each iteration. Wrapper methods, while computationally more intensive than filter methods, are advantageous for capturing feature interactions and dependencies that contribute to optimal model performance. However, their increased computational cost may limit their application to high-dimensional datasets. Embedded methods for feature selection incorporate feature selection as part of the model training process. Unlike filter methods, which assess features independently of the learning algorithm, and wrapper methods, which evaluate subsets of features through iterative model training, embedded methods simultaneously perform feature selection and model training. These methods aim to identify the most relevant features for prediction and classification tasks while optimizing the model's performance. One popular embedded method is Least Absolute Shrinkage and Selection Operator, which introduces a penalty term to the linear regression cost function, promoting sparsity in the feature coefficients. Tree-based algorithms like Random Forests and Gradient Boosted Trees also inherently provide feature importance scores during their training process, allowing for the automatic selection of the most influential features. Embedded methods are advantageous as they streamline the feature selection process within the model training, potentially leading to more efficient and interpretable models. Dimensionality reduction techniques are methods employed in ML to reduce the number of input features while preserving the essential information within the data. One widely used technique is Principal Component Analysis, which transforms the original features into a set of uncorrelated variables called principal components. These components retain most of the variance in the data, enabling a more compact representation. Hybrid methods in feature selection represent a fusion of multiple techniques to achieve a more comprehensive and robust approach. These methods combine aspects of both filter and wrapper methods or leverage various strategies simultaneously. For instance, Boruta integrates the power of random forest classifiers with a shadow feature mechanism to identify relevant features, providing a hybrid solution. Genetic Algorithms, another hybrid approach, employs evolutionary algorithms to search for an optimal subset of features. Hybrid methods strive to harness the strengths of different feature selection techniques, addressing their limitations and producing more effective results. By combining diverse strategies, these methods offer a versatile and adaptable approach to feature selection, suitable for various datasets and ML tasks. The choice of a hybrid method depends on the specific characteristics of the data and the goals of the feature selection process. Each type of feature selection caters to specific data characteristics and model requirements, which is crucial in optimizing performance and interpretability in ML applications [7], [63]. In this research,

we applied the embedded method because it is faster and less computationally expensive than other methods and is suitable for ML models.

5.1.5 Balancing Data sets

Balancing data sets is an essential step in ML and data analysis when dealing with imbalanced data, where the number of instances in different classes or categories is significantly skewed[13], [14]. Balancing the data sets helps ensure that the model's performance is not biased towards the majority class and can effectively learn from the minority class. In practice, the datasets of software bugs and code smell often suffer from a common problem which is a class imbalance problem[40]. The reference datasets are not balance distributed, which shows a lack in the actual distribution of learning instances (The number of defective or smelly cases is smaller than non-defective or non-smelly), we manage this problem by modifying the original datasets to increase the realism of the data. The distribution of the dataset was modified by applying different data sampling methods such as Near Miss, Tomek links, Random Oversampling, SMOTE, and SMOTE Tomek.

- The process of Near Miss is as follows:

1- Identify minority class instances: Identify the instances belonging to the minority class.
2- Near Miss Selection:

- For each instance in the minority class, calculate the distance to its k nearest neighbors in the majority class. The instances in the majority class that are closest to the minority class form "near misses."
- Select the "near misses" based on a criterion. There are three common types of Near Miss methods:
  - Near Miss-1: Keep majority instances whose average distance to k nearest minority instances is the smallest.
  - Near Miss-2: Keep majority instances whose average distance to k nearest minority instances is the largest.
  - Near Miss-3: Remove majority instances if the average distance to k nearest minority instances is smaller than the average distance to k nearest majority instances.

3- Majority class reduction: Remove the selected majority class instances to balance the class distribution. This reduction process aims to create a balanced dataset with fewer instances from the majority class.
4- Balanced dataset: Combine the minority class instances with the selected majority class instances to create a balanced dataset.

- The process of Tomek Links is as follows:

1- Identify minority class instances: Identify the instances belonging to the minority class (fraudulent transactions).
2- Find Nearest Neighbors:

- Calculate the distance to all other instances for each instance in the dataset.
- For each instance, identify its nearest neighbor from a different class. A Tomek link is formed if:
  - Instance A belongs to the minority class.
  - Instance B belongs to the majority class.
  - Instance B is the nearest neighbor of instance A.
  - Instance A is the nearest neighbor of instance B.

3- Tomek Link removal: Remove the instances that form Tomek links. This process removes instances that are ambiguous or near the decision boundary between classes.
4- Balanced dataset: Combine the instances after removing Tomek links to create a more balanced dataset.

- The process of Random Oversampling is as follows:

1- Identify minority class instances: Identify the instances belonging to the minority class.
2- Random Oversampling: Randomly select instances from the minority class and duplicate them until the desired proportion of the minority class is met.
3- Repeat the process: Repeat step 2 until the class distribution is balanced. The number of duplicates needed depends on the degree of imbalance and the desired balance ratio.
4- Balanced dataset: Combine the original minority class instances with duplicated ones to create a more balanced dataset.

- The process to generate the synthetic samples SMOTE is as follows:

1- Choose random data from the minority class.
2- Calculate the Euclidean distance between the random data and its k nearest neighbors.
3- Multiply the difference with a random number between 0 and 1, then add the result to the minority class as a synthetic sample.
4- Repeat the procedure until the desired proportion of minority class is met.

- The process of SMOTE-Tomek is as follows:

1- (Start of SMOTE) Choose random data from the minority class.
2- Calculate the distance between the random data and its k nearest neighbors.
3- Multiply the difference with a random number between 0 and 1, then add the result to the minority class as a synthetic sample.
4- Repeat step number 2–3 until the desired proportion of minority class is met. (End of SMOTE)
5- (Start of Tomek Links) Choose random data from the majority class.
6- If the random data nearest neighbor is the data from the minority class (i.e. create the Tomek Link), then remove the Tomek Link.

Figures 5.2 to 5.7 show the distribution of learning instances over the original and balanced data sets.

- Regarding the unified bug dataset: The distribution of learning defective instances over the original data sets (Class level metrics and File level metrics) is (8780 and 10240), respectively. At the same time the distribution of learning non-defective instances is (38838 and 33504), respectively.
- ➢ Following the implementation of the Near Miss method, the distribution of learning defective instances over the balanced data sets (Class level metrics and File level metrics) became (8780 and 10240), respectively. While the distribution of learning non-defective instances became (8780 and 10240), respectively.

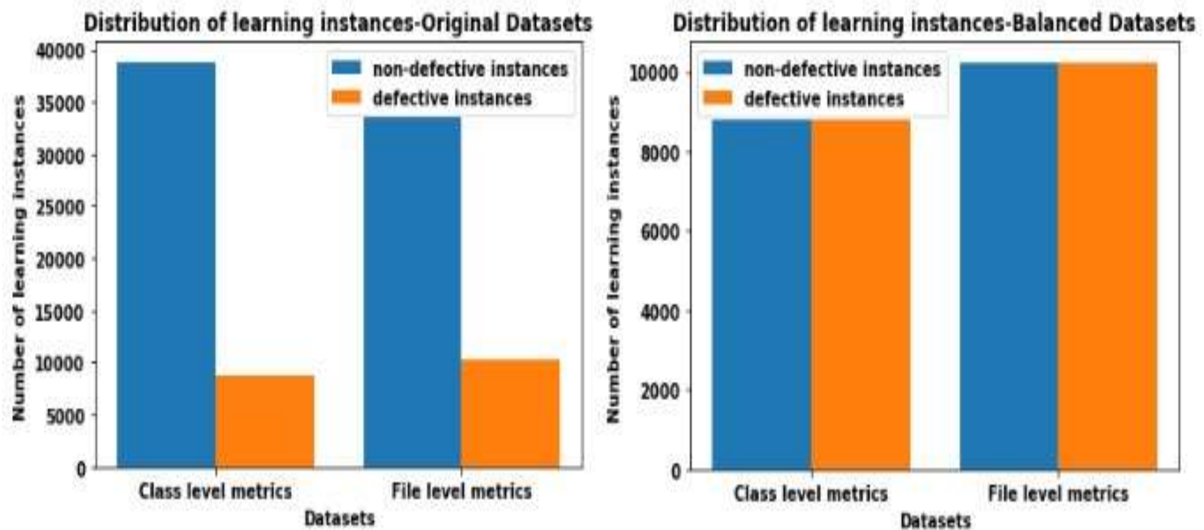


Figure 5.2 Distribution of learning instances over the original and balanced data sets (The public unified bug dataset)-by applying the Near Miss method

- Regarding PROMISE datasets: The distribution of learning defective instances over the original data sets (ant, camel, ivy, jedit, log4j, and xerces) is (166, 188, 40, 11, 16, and 151), respectively. At the same time the distribution of learning non-defective instances is (579, 777, 312, 481, 189, and 437), respectively.

➢ Following the implementation of SMOTE Tomek method, the distribution of learning defective instances over the balanced data sets (ant, camel, ivy, jedit, log4j, and xerces) became (559, 751, 297, 466, 185 and 418), respectively. While the distribution of learning non-defective instances became (559, 751, 297, 466, 185 and 418), respectively.

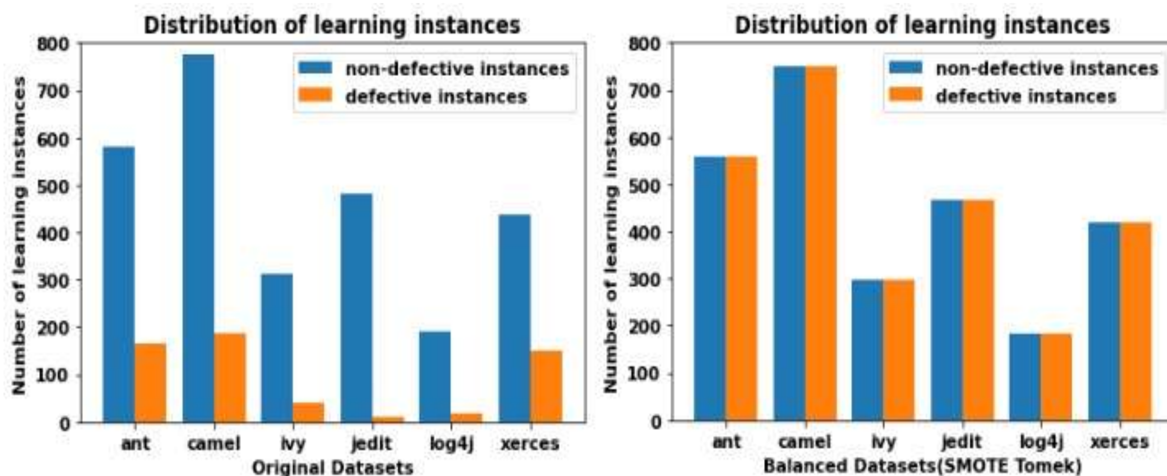


Figure 5.3 Distribution of learning instances over the original and balanced data sets (The PROMISE datasets)-by applying the SMOTE Tomek method

➢ Following the implementation of Random Oversampling method, the distribution of learning defective instances over the balanced data sets (ant, camel, ivy, jedit, log4j, and xerces) became (579, 777, 312, 481, 189 and 437), respectively. At the same time the distribution of learning non-defective instances became (579, 777, 312, 481, 189, and 437), respectively.

- ➢ Following the implementation of SMOTE method, the distribution of learning defective instances over the balanced data sets (ant, camel, ivy, jedit, log4j, and xerces) became (579, 777, 312, 481, 189 and 437), respectively. While the distribution of learning non-defective instances became (579, 777, 312, 481, 189, and 437), respectively.

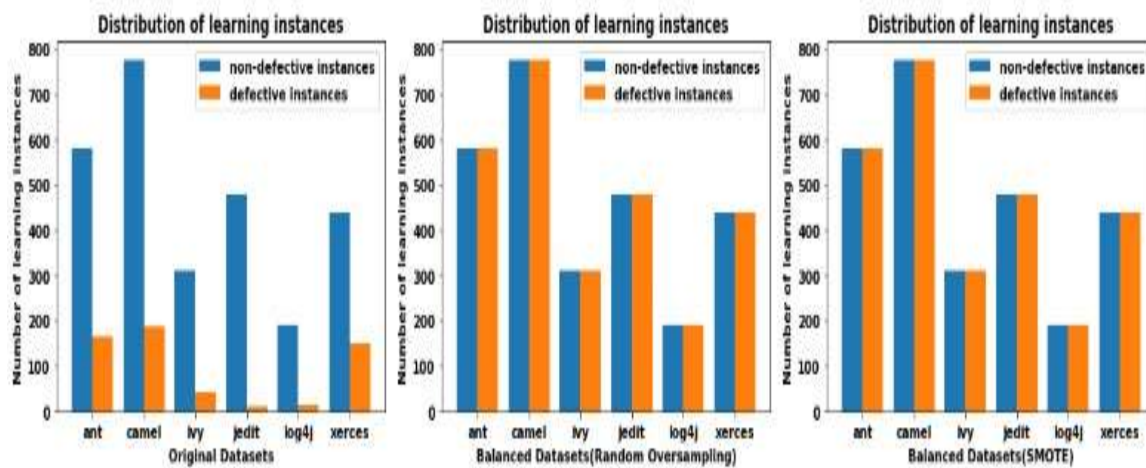


Figure 5.4 Distribution of learning instances over the original and balanced data sets (The PROMISE datasets)-by applying the Random Oversampling and SMOTE methods

- Regarding the QC systems datasets: The distribution of learning smelly instances over the original data sets (God Class, Data Class, Feature envy and Long method) is (140, 140, 140 and 141), respectively. At the same time the distribution of learning non-smelly instances is (280, 280, 280 and 279), respectively.
- ➢ Following the implementation of SMOTE method, the distribution of learning smelly instances over the balanced data sets (God Class, Data Class, Feature envy and Long method) became (280, 280, 280 and 279), respectively. While the distribution of learning non-smelly instances became (280, 280, 280 and 279), respectively.

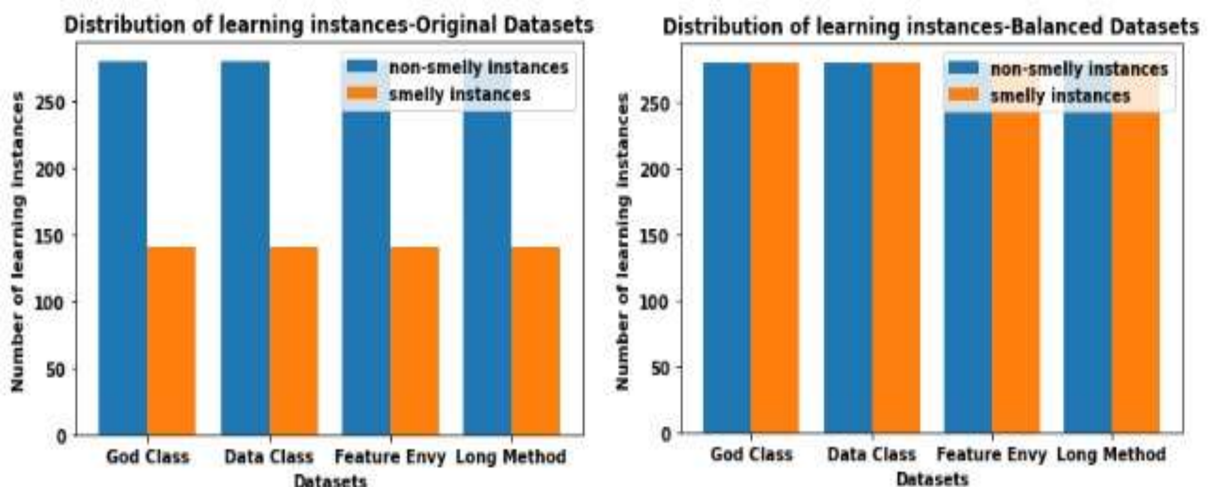


Figure 5.5 Distribution of learning instances over the original and balanced data sets (The Qualitas Corpus Systems)-by applying the SMOTE method

- ➢ Following the implementation of Random Oversampling method, the distribution of learning smelly instances over the balanced data sets (God Class, Data Class, Feature envy

and Long method) became (280, 280, 280 and 279), respectively. At the same time the distribution of learning non-smelly instances became (280, 280, 280 and 279), respectively.

- Following the implementation of Tomek Links method, the distribution of learning smelly instances over the balanced data sets (God Class, Data Class, Feature envy and Long method) became (140, 140, 140 and 141), respectively. While the distribution of learning non-smelly instances became (263, 256, 261and 270), respectively.

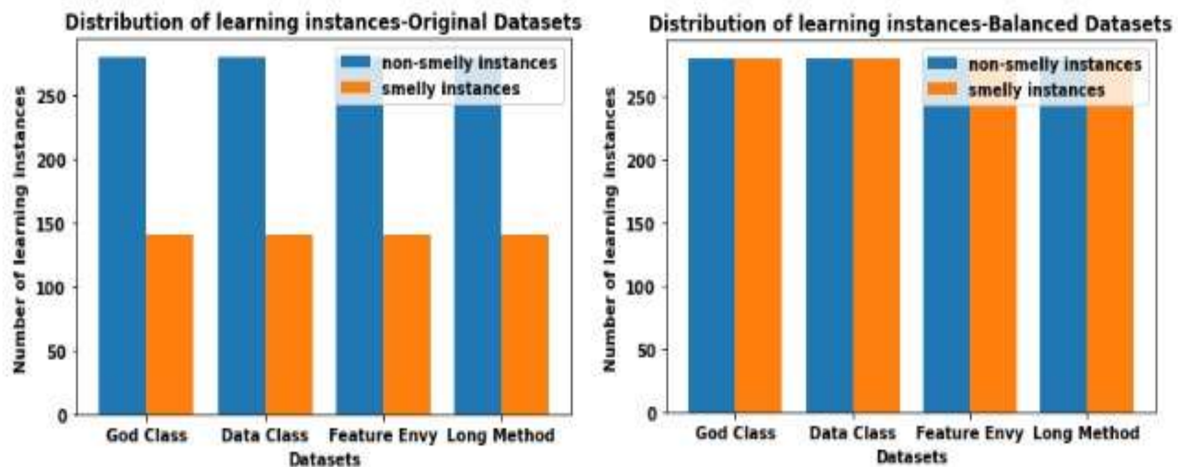


Figure 5.6 Distribution of learning instances over the original and balanced data sets (The Qualitas Corpus Systems)-by applying the Random Oversampling method

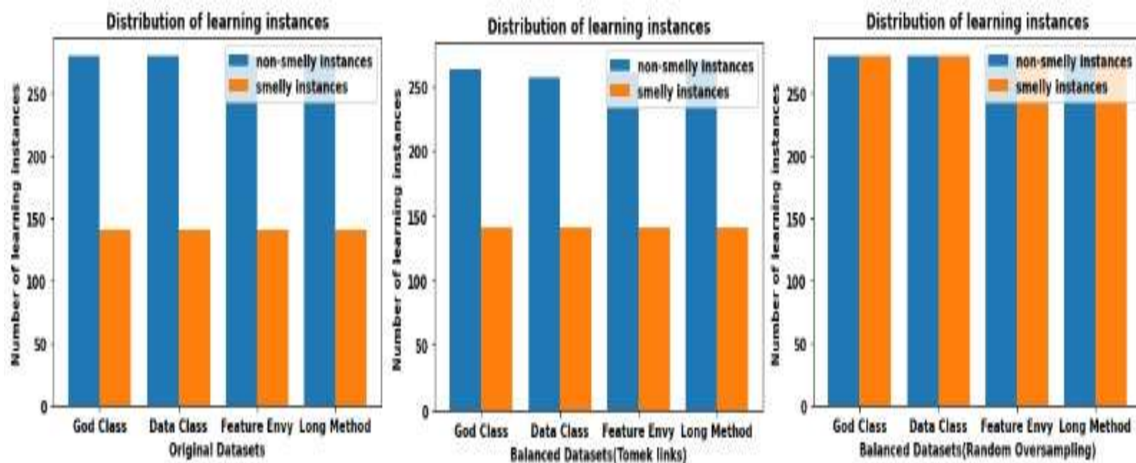


Figure 5.7 Distribution of learning instances over the original and balanced data sets (The Qualitas Corpus Systems)-by applying the Random Oversampling and Tomek Links methods

5.1.6 Models Building and Evaluation

In building and evaluating the proposed prediction models, we adopted a systematic and methodical methodology which depends on ML techniques in conjunction with data-balancing methods to predict software bugs and code smells effectively. It's a common practice in the field to divide data into two sets: a training set used to teach the model and a test set used to assess its performance [112]. The datasets used to train and test our proposed ML models were obtained from public benchmark datasets of software bugs and code smells that contain information for several projects. Datasets are shuffled and split into testing and training sets. Training is performed with 80% of the dataset (random selection of features), while the

remaining 20% is used for validation and testing. The author utilized the Jupyter editor as a computing environment to construct models using the Python programming language to implement the methodology. Moreover, we harnessed a range of libraries and tools to efficiently handle data, construct models, and create insightful visualizations. Specifically, Pandas for data manipulation, scikit-learn, Keras, and TensorFlow for data modeling, and Matplotlib along with Seaborn for data visualization were employed. Moreover, Cross-validation is a vital technique in ML used to evaluate the performance and generalizability of predictive models. It involves partitioning a dataset into subsets, typically referred to as folds, and systematically training and evaluating the model multiple times. Cross-validation helps mitigate issues like overfitting and provides a more reliable assessment of how well a model will perform on unseen data. It is an essential tool for selecting models, tuning hyperparameters, and ensuring the model's generalization across different subsets of the dataset. Cross-validation comes in various forms such as K-Fold Cross-Validation, Stratified K-Fold Cross-Validation, Leave-One-Out Cross-Validation, Leave-P-Out Cross-Validation, etc. to suit different data characteristics and modelling objectives. K-Fold Cross-Validation and Stratified K-Fold Cross-Validation are the most standard methods of Cross-validation. K-Fold Cross-Validation is a method where the data is divided into k subsets, and the model is trained on k-1 folds while being tested on the remaining fold. This process is repeated k times, and performance metrics are averaged to provide a more robust estimate of the model's effectiveness. Stratified K-Fold Cross-Validation is a variation of the standard K-Fold Cross-Validation method that maintains the class distribution in each fold, is beneficial for imbalanced datasets, and is designed to address the potential issue of imbalanced class distributions in the dataset. Therefore, we applied Stratified K-Fold Cross-Validation method to evaluate the performance of our proposed predictive models. Each model was developed separately with different parameters. Once a prediction model is built, its performance must be evaluated. We evaluated the performance of our proposed models based on a set of standard performance measures such as the confusion matrix, Matthews Correlation Coefficient (MCC), the area under a receiver operating characteristic curve (AUC), the area under the precision-recall curve (AUCPR) and mean square error (MSE) [17], [39].

- Parameter settings of the models:

Hyperparameters encompass a diverse set of configuration settings crucial for shaping the behaviour of ML models. For instance, in Support Vector Machines, Kernel Parameters, such as those in the Radial Basis Function, significantly influence the model's capacity to handle complex relationships in the data. Decision Tree Parameters, including maximum depth and minimum samples per split, are pivotal for controlling the tree's complexity and preventing overfitting. Random Forests involve hyperparameters like the Number of Trees and Depth, determining the ensemble's robustness and individual tree characteristics. In k-Nearest Neighbors, the choice of k, or the number of nearest neighbors considered, impacts the model's flexibility and sensitivity to noise [39]. Additionally, Neural Networks involve several hyperparameters, such as Cell Type, Bidirectional layers, Dropouts, Dense layers, Optimizer, Learning Rate, Regularization Strength, Number of Iterations (Epochs), Batch Size, Hidden Layers, and Neurons, each playing a role in the network's architecture, convergence, and generalization. Learning Rate, a critical hyperparameter, dictates the step size during optimization, affecting the convergence speed and potential overshooting of optimal solutions. Regularization Strength is pivotal for preventing overfitting by controlling the complexity of

the model. The Number of Iterations (Epochs) determines how many times the entire training dataset is processed, balancing between underfitting and overfitting. Batch Size influences the optimization efficiency, impacting both speed and memory usage. Several Hidden Layers and Neurons, pivotal for capturing intricate relationships within data. Activation Functions introduce non-linearity, influencing the model's capacity to learn intricate mappings. Practical tuning of these hyperparameters is essential for optimizing model performance across diverse ML paradigms. Tables 5.5 and 5.6 show the parameter settings of the models [17], [37].

Table 5.5 Parameter settings of the models (Classical techniques)

Models	parameters
NB	No passing parameters (default parameters)
LR	Random_state=0
DT	No passing parameters (default parameters)
RF	n_estimators = 100
K-NN	n_neighbors = 7
SVM	probability = True, kernel = 'linear'
XGB	max_depth=3, n_estimators=100, n_jobs=2, objectvie='binary:logistic', learning_rate=0.01, subsample=0.7, colsample_bytree=0.8
MLP	hidden_layer_sizes=(10,5), max_iter=1000

Table 5.6 Parameter settings of the models (Advanced techniques)

Parameters	Models			
	Bi-LSTM	LSTM	CNN	GRU
Cell type (Bidirectional)	LSTM (64, 32), return_sequences =True	LSTM (64, 32), return_sequences =True	-	-
Layers. GRU	-	-	-	100
Activation function	ReLU + sigmoid	ReLU + sigmoid	ReLU + Sigmoid	Tanh + Sigmoid
Dropouts	0.2	0.2	0.2	0.2
Dense	64, 1	64, 1	10, 1	1
Optimizer	Adam	Adam	Adam	Adam
Learning Rate	0.01	0.01	0.01	0.01
Loos Function	Mean squared error (MSE)	Mean squared error (MSE)	Mean squared error (MSE)	Mean squared error (MSE)
Batch Size	64	64	25	64
Epochs	100	100	100	100
Validation Split	0.1	0.1	0.1	0.1
Verbose	1	1	-	1

- A *confusion matrix* is a specific Table used to measure the performance of a model. Accuracy, Precision, Recall, and F-measure are the typical performance measurement parameters used in the confusion matrix. A confusion matrix summarizes the results of the testing algorithm. It presents a report of (i) True Positive Rate (*TPR*), (ii) False Positive Rate (*FPR*), (iii) True Negative Rate (*TNR*), and (iv) False Negative Rate (*FNR*)[18], [112]. Table 5.7 shows the confusion matrix.

Table 5.7 Confusion matrix

Predicted Values	Actual Values	
	Positive (Yes)	Negative (No)
Positive (Yes)	TP	FP
Negative (No)	FN	TN

- The accuracy is the ratio of true results that are calculated as the sum of true positive and true negative instances divided by the sum of true positive, true negative, false positive and

false negative. The top (maximum) accuracy is 1, whereas the low (minimum) accuracy is 0[18]. Accuracy can be computed by using the following formula:

$$\text{Accuracy} = \frac{(\text{TP} + \text{TN})}{(\text{TP} + \text{TN} + \text{FP} + \text{FN})} \quad (24)$$

- Precision is defined as the number of true positive predictions divided by the total number of positive predictions or fraction of true positive and predicted yes instances[18]. The top (maximum) precision is 1, whereas the low (minimum) is 0 and it can be calculated as:

$$\text{Precision} = \frac{\text{TP}}{(\text{TP} + \text{FP})} \quad (25)$$

- The recall is the number of positive predictions divided by the total number of positives or defined as the fraction between true positive instances and actual yes instances. The top (maximum) recall is 1, whereas the low (minimum) is 0[18]. The formula of recall is given below:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (26)$$

- The F-Measure is the weighted harmonic mean of precision and recall or defined as the fraction between the product of the recall and precision to the summation of recall and precision parameter of classification, it is used to combine the recall and precision measures in one measure to compare different algorithms[18]. The F-Measure formula is given below:

$$\text{F} - \text{Measure} = \frac{(2 * \text{Recall} * \text{Precision})}{(\text{Recall} + \text{Precision})} \quad (27)$$

- The Matthews Correlation Coefficient (MCC) is a measure used for model evaluation by measuring the difference between the predicted values and actual values [81], [82], [101].The MCC formula is given below:

$$\text{MCC} = \text{TP} * \text{TN} - \text{FP} * \text{FN} \,/\, \sqrt{(\text{TP} + \text{FP}) * (\text{TP} + \text{FN}) * (\text{TN} + \text{FP}) * (\text{TN} + \text{FN})} \quad (28)$$

- The Area Under the ROC Curve (AUC) is a graph that shows the performance of classification models with all classification thresholds and plots based on two parameters, actual positive rate (*TPR*) and false-positive rate (*FPR*) [61], [112]. The AUC formula is given below:

$$\text{AUC} = \frac{\sum_{\text{ins}_i \in \text{Positive Class}} \text{rank}(\text{ins}_i) - \frac{\text{M}(\text{M}+1)}{2}}{\text{M} \,.\, \text{N}} \quad (29)$$

Where $\sum_{ins_i \in Positive\ Class} rank(ins_i)$ It is the sum of the ranks of all positive samples, and *M* and *N* are the number of positive and negative examples, respectively.

- The Area Under the Precision-Recall curve (AUCPR) is a curve that plots the Precision versus the Recall or a single number summary of the information in the precision-recall curve[113]. The AUCPR formula is given below:

$$\text{AUCPR} = \int_0^1 \text{Precision}(\text{Recall}) \, d(\text{Recall}) \quad (30)$$

- The Mean Square Error (MSE) is a metric that measures the amount of error in the model. It assesses the average squared difference between the actual and predicted values [42], [112]. The MSE formula is given below:

$$\mathrm{MSE} = \frac{1}{n} \sum_{i=1}^{n} (x(i) - y(i))^2 \tag{31}$$

Where n is the number of observations, $x(i)$ is the actual value, $y(i)$ is the observed or predicted value for the ith observation.

5.2 Summary

This chapter presents the proposed methodology and implementation for predicting software bugs and code smells. Our proposed methodology was based on various ML techniques and data-balancing methods (data sampling methods). Public benchmark datasets of software bugs and code smells have been used to ensure the methodology performs well across different types of software projects. To check how well our methodology works, we balanced the original data sets using different data sampling methods and then conducted extensive Python experiments. Additionally, we used various Hyperparameters to set our proposed models and evaluate the model's performance using various performance measures.

Chapter 6 Experimental Results and Discussion of Software Bugs Prediction (SBP)

This subsection presents the results obtained from the experiments explained in the previous section (proposed methodology and implementation) which includes the results of SBP.

6.1 ML Techniques in SBP

In this sub-section, we discuss the findings of the first study. The goal was to present a comprehensive study on ML techniques successfully used in previous studies to predict software bugs. The study also presented a method for SBP based on supervised ML algorithms namely, DT, NB, RF, and LR. The experiments have been conducted based on benchmark datasets obtained from the NASA datasets (jm1, PC1, KC1 and KC2). The experimental results were evaluated and compared based on various performance measures (accuracy, precision, recall, f-measure, and AUC).

The performance of the prediction models is reported in Tables 6.1 to 6.6 and Figures 6.1 to 6.4.

Tables 6.1 to 6.4 show the performance of the proposed models on the four data sets based on all performance measures. The maximum (best) accuracy value is 99%, which DT and RF models in JM1, PC1and KC1 datasets achieved. The maximum (best) precision value is 99%, which DT and RF models in JM1, PC1and KC1 datasets achieved. The maximum (best) recall value is 100%, which was achieved by DT and RF models in all datasets. The maximum (best) F-measure value is 99%, achieved by DT and RF models in the PC1 dataset.

Table 6.1 Performance measures of the proposed models on the jm1 dataset

Proposed models	Performance measures			
	Accuracy	Precision	Recall	F-measure
DT	0.99	0.99	1.00	0.99
NB	0.80	0.81	0.97	0.89
RF	0.99	0.99	1.00	0.99
LR	0.81	0.82	0.99	0.89

Table 6.2 Performance measures of the proposed models on the pc1 dataset

Proposed models	Performance measures			
	Accuracy	Precision	Recall	F-measure
DT	0.99	0.99	1.00	1.00
NB	0.91	0.94	0.96	0.95
RF	0.99	0.99	1.00	1.00
LR	0.93	0.94	0.99	0.96

Table 6.3 Performance measures of the proposed models on the kc1 dataset

Proposed models	Performance measures			
	Accuracy	Precision	Recall	F-measure
DT	0.99	0.99	1.00	0.99
NB	0.85	0.88	0.96	0.92
RF	0.99	0.99	1.00	0.99
LR	0.85	0.87	0.96	0.92

Table 6.4 Performance measures of the proposed models on the kc2 dataset

Proposed models	Performance measures			
	Accuracy	Precision	Recall	F-measure
DT	0.98	0.98	1.00	0.99
NB	0.83	0.83	0.98	0.90
RF	0.98	0.98	1.00	0.99
LR	0.84	0.86	0.96	0.91

Figures 6.1 to 6.4 present the Receiver Operating Characteristic (ROC) Curves for the proposed models on the four data sets. The vertical axis presents the actual positive rate of the model, and the horizontal axis illustrates the false positive rate. The AUC is a sign of the performance of the model. The larger AUC is, the better the model performance will be. Based on the Figures, the values are encouraging and indicate our proposed model's efficiency in SBP. Regarding the *jm1* dataset, the best AUC is 97%, which the DT and RF models obtain. The worst AUC is 52% which is obtained by the NB and LR models. Regarding the *pc1* dataset, the best AUC is 96% which the DT and RF models obtain. The worst AUC is 54%, which the NB model obtains. Regarding the *kc1* dataset, the best AUC is 96% which the DT and RF models obtain. The worst AUC is 59%, which the LR model obtains. Regarding the kc2 dataset, the best AUC is 96%, which the DT and RF models obtain. The worst AUC is 60%, which the NB model obtains. The results show that DT and RF models have better AUC values than NB and LR models.

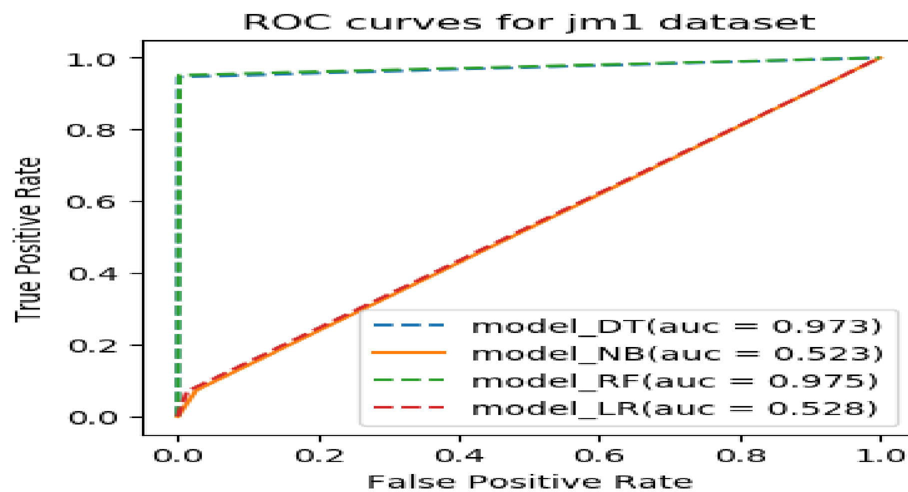


Figure 6.1 Comparison of ROC curves for Models Across the jm1 Dataset

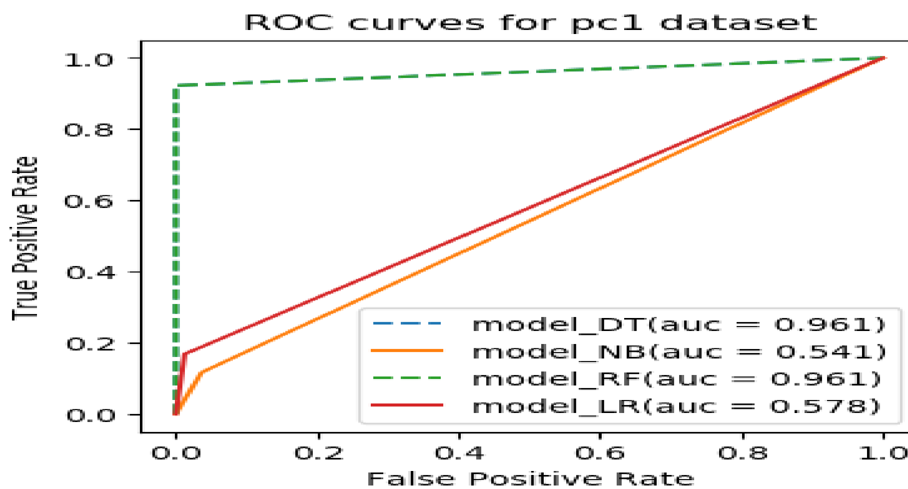


Figure 6.2 Comparison of ROC curves for Models Across the pc1 Dataset

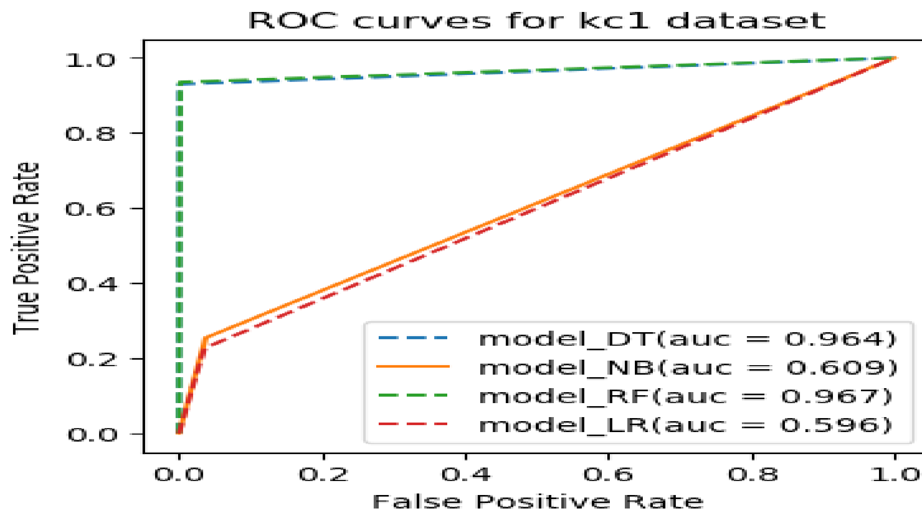


Figure 6.3 Comparison of ROC curves for Models Across the kc1 Dataset

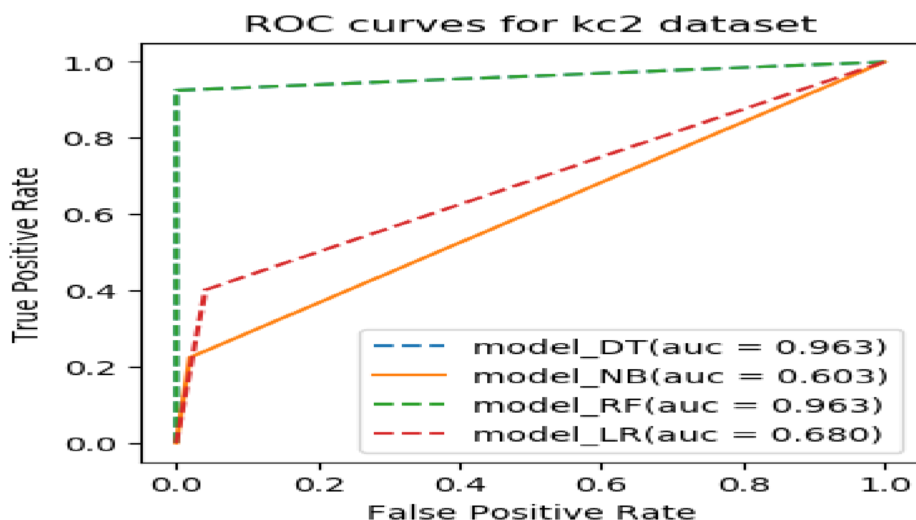


Figure 6.4 Comparison of ROC curves for Models Across the kc2 Dataset

Tables 6.5 and 6.6 show our study's comparison results with previous studies that used the same dataset based on performance measures, namely accuracy, precision, recall and f-measure. The best values are indicated with bold text and "- "to indicate the approaches that did not provide results in a particular data set. According to the Tables, some of the results in the previous studies are better than ours. Still, in most cases, our method outperforms the other state-of-the-art methods and provides better predictive performance.

Table 6.5 Comparing the results of our study with the results of studies that used the same dataset and algorithms across the jm1 and pc1 dataset

jm1 dataset					
Performance measure	**ML models**	**Studies**			
		First study[82]	**Second study**[114]	**Third study**[10]	**Our study**
Accuracy	DT	-	-	0.81	**0.99**
	NB	-	-	0.81	0.80
	RF	-	-	0.82	**0.99**
F-measure	DT	-	-	0.90	**0.99**
	NB	0.75	-	0.89	0.89
	RF	0.76	-	0.90	**0.99**

	LR	0.74	-	-	0.89
pc1 dataset					
Accuracy	DT	-	-	0.93	**0.99**
	NB	-	-	0.88	0.91
	RF	-	-	0.93	**0.99**
F-measure	DT	-	-	0.97	**1.00**
	NB	0.89	-	0.94	0.95
	RF	0.91	-	0.97	**1.00**
	LR	0.91	-	-	0.96

Table 6.6 Comparing the results of our study with the results of studies that used the same dataset and algorithms across the kc1 and kc2 datasets

kc1 dataset					
Performance measure	**ML models**	**Studies**			
		First study[82]	**Second study**[114]	**Third study**[10]	**Our study**
Accuracy	DT	-	-	0.84	**0.99**
	NB	-	0.82	0.82	0.85
	RF	-	-	0.85	**0.99**
Precision	NB	-	0.80	-	**0.88**
Recall	NB	-	0.83	-	**0.96**
F-measure	DT	-	-	0.92	**0.99**
	NB	0.82	0.81	0.90	0.92
	RF	0.82	-	0.92	**0.99**
	LR	0.81	-	-	0.92
kc2 dataset					
Accuracy	DT	-	-	0.82	**0.98**
	NB	-	-	0.84	0.83
	RF	-	-	0.82	**0.98**
F-measure	DT	-	-	0.89	**0.99**
	NB	0.80	-	0.90	0.90
	RF	0.76	-	0.89	**0.99**
	LR	0.79	-	-	0.91

In summary, this research aimed to provide a comprehensive study on ML techniques in SBP, and propose a method for SBP based on supervised ML algorithms. The results of the proposed method were compared with some results presented in previous studies. When conducting the literature review, we uncovered many papers delving into the realm of ML models for predicting software bugs. Notably, our examination revealed that a predominant portion of these studies employed ML techniques such as NB, ANNs and SVM for software bug prediction. Additionally, it is worth noting that our review encompasses an array of research papers, each contributing unique insights into the application and effectiveness of these ML models in the context of bug prediction. Regarding evaluating the results obtained from our proposed method and their comparison with the results of other studies, we conclude that the DT and RF classifiers achieved commendable scores compared to other classifiers, and our method outperforms other methods in predicting software bugs. The evaluation process and the study's results unequivocally demonstrate the efficacy of ML algorithms in SBP. Furthermore, this research underscores the need for additional investigation into the realm of static code analysis, as it can potentially uncover and predect software bugs more comprehensively. In our future work, we will combine ML techniques with data-balancing method to improve the accuracy of SBP. By employing ML techniques coupled with advanced data-balancing methods, we can not only enhance the accuracy of SBP but also pave the way for more robust and reliable software development practices.

6.2 LSTM and GRU with Undersampling Methods in SBP

In this sub-section, we discuss the findings of the second study. The goal was to present a method based on combining two RNN models namely LSTM and GRU with the Undersampling method (Near Miss) for SBP. The experiments have been conducted based on benchmark datasets obtained from the public unified bug dataset. The experimental results were evaluated and compared based on various performance measures (accuracy, precision, recall, f-measure, MCC, AUC, AUCPR and MSE).

The performance of the prediction models is reported in Tables 6.7 to 6.9, and Figures 6.5 to 6.11, appendix 1 (Figures 1 and 2).

Table 6.7 shows the results of the LSTM and GRU models based on both the original and balanced datasets, emphasising class-level measures.. Notably, we observed that both the LSTM and GRU models attained the highest accuracy of 93% on the balanced dataset, while the GRU model exhibited the lowest accuracy of 82% on the original dataset. In terms of precision, the LSTM model achieved the highest value of 95% on the balanced dataset, while the GRU model demonstrated the lowest precision of 58% on the original dataset. As for recall, both models obtained the highest score of 92% on the balanced dataset, whereas the GRU model exhibited the lowest recall of 16% on the original dataset. Both models achieved the highest F-Measure score of 93% on the balanced dataset. However, the GRU model had the lowest score of 26% on the original dataset. . Both models achieved the highest MCC of 86% on the balanced dataset, whereas the GRU model had the lowest MCC of 23% on the original dataset. The LSTM model attained the highest AUC score of 97% on the balanced dataset, and the GRU model achieved the lowest score of 77% on the original dataset. On the balanced dataset, both models demonstrated the highest AUCPR score of 97%, while the GRU model exhibited the lowest AUCPR score of 44% on the original dataset. Additionally, the GRU model recorded the highest MSE of 0.130 on the original dataset, while the LSTM model achieved the lowest MSE of 0.051 on the balanced dataset.

Table 6.7 Performance measures for the proposed models over class level metrics dataset

Original Dataset								
Proposed Models	**Performance Measures**							
	Accuracy	**Precision**	**Recall**	**F-measure**	**MCC**	**AUC**	**AUCPR**	**MSE**
LSTM	0.83	0.60	0.25	0.35	0.30	0.78	0.48	0.125
GRU	0.82	0.58	0.16	0.26	0.23	0.77	0.44	0.130
Averages	**0.82**	**0.59**	**0.20**	**0.30**	**0.26**	**0.77**	**0.46**	**0.130**
Balanced Dataset								
Proposed Models	**Performance Measures**							
	Accuracy	**Precision**	**Recall**	**F-measure**	**MCC**	**AUC**	**AUCPR**	**MSE**
LSTM	0.93	0.95	0.92	0.93	0.86	0.97	0.97	0.051
GRU	0.93	0.94	0.92	0.93	0.86	0.96	0.97	0.063
Averages	**0.93**	**0.94**	**0.92**	**0.93**	**0.86**	**0.96**	**0.97**	**0.057**

Table 6.8 shows the results of LSTM and GRU models based on on the original and balanced datasets, focusing on file-level metrics. Remarkably, both the LSTM and GRU models achieved the highest accuracy of 88% on the balanced dataset. In contrast the lowest accuracy of 78% was observed for both models (LSTM and GRU) on the original dataset. Furthermore, the balanced dataset yielded the highest precision of 94% for both models (LSTM and GRU), while the GRU model had the lowest precision of 61% on the original dataset. Regarding recall, the balanced dataset produced the highest score of 81% for both models. Conversely, when applied to the original dataset, the LSTM model achieved the lowest recall of 18%. Similarly,

the balanced dataset resulted in the highest f-measure of 87% for both the LSTM and GRU models. Conversely, the LSTM model exhibited the lowest f-measure of 28% when working with the original dataset. Furthermore, both models (LSTM and GRU) attained the highest MCC of 76% on the balanced dataset, while the LSTM model had the lowest MCC of 24% on the original dataset. Similarly, the balanced dataset yielded the highest AUC of 93% for both models (LSTM and GRU), while the original dataset yielded the lowest AUC of 75% for both models (LSTM and GRU). Both models also achieved the highest AUCPR on the balanced dataset, 95%, and the lowest AUCPR on the original dataset, 49%. In conclusion, both models (LSTM and GRU) achieved the highest MSE of 0.152 on the original dataset, while the LSTM model obtained the lowest MSE of 0.090 on the balanced dataset.

Table 6.8 Performance measures for the proposed models over file level metrics dataset

Original Dataset								
Proposed Models	**Performance Measures**							
	Accuracy	**Precision**	**Recall**	**F-measure**	**MCC**	**AUC**	**AUCPR**	**MSE**
LSTM	0.78	0.62	0.18	0.28	0.24	0.75	0.49	0.152
GRU	0.78	0.61	0.22	0.33	0.27	0.75	0.49	0.152
Averages	**0.78**	**0.61**	**0.20**	**0.30**	**0.25**	**0.75**	**0.49**	**0.152**
Balanced Dataset								
Proposed Models	**Performance Measures**							
	Accuracy	**Precision**	**Recall**	**F-measure**	**MCC**	**AUC**	**AUCPR**	**MSE**
LSTM	0.88	0.94	0.81	0.87	0.76	0.93	0.95	0.090
GRU	0.88	0.94	0.81	0.87	0.76	0.93	0.95	0.093
Averages	**0.88**	**0.94**	**0.81**	**0.87**	**0.76**	**0.93**	**0.95**	**0.091**

Boxplots are particularly useful for comparing distributions between group or visualizing multiple datasets or subsets within a single dataset. Therefore, we aggregated the achieved results to get a more accurate overview of the quality of the results using boxplots. Figure 6.5 displays Box plots, which effectively depict a ranges of performance measures for all datasets. The ranges of performance measures (Accuracy, Precision, Recall, F-measure, MCC, AUC and AUCPR) on the original datasets are 78% to 83%, 58% to 62%, 16% to 25%, 26% to 35%, 23% to 30%, 75% to 78%, 44% to 49%, respectively. While, the ranges of performance measures (Accuracy, Precision, Recall, F-measure, MCC, AUC and AUCPR) on the balanced datasets are 88% to 93%, 94% to 95%, 81% to 92%, 87% to 93%, 76% to 86%, 93% to 97%, 95% to 97%, respectively.

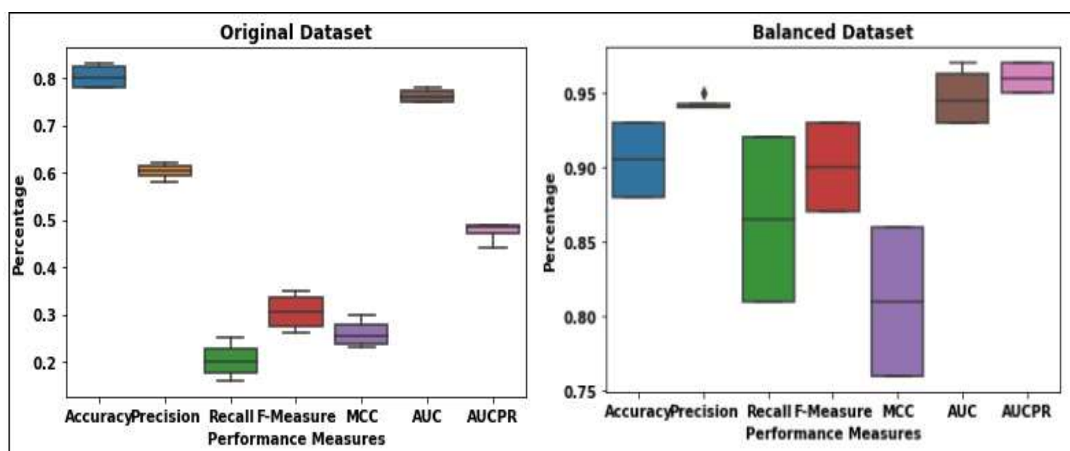


Figure 6.5 Showcases the boxplots illustrating the performance measures achieved by the proposed models on all datasets, encompassing both class-level and file-level metrics

Figures 6.6 to 6.9 show the training and validation accuracy and training and validation loss of the models on the original and balanced datasets.
Figures 6.6 and 6.7 show the training and validation accuracy of the models on the original and balanced datasets. The vertical axis presents the accuracy of the model, and the horizontal axis illustrates the number of epochs. Accuracy is the fraction of predictions that our model predicted right.
Regarding the original datasets, the LSTM model learned 83% accuracy for the class-level metrics dataset and 78% accuracy for the file level metrics dataset dataset at the 100th epoch. The GRU model learned 82% accuracy for the class level metrics dataset and 78% accuracy for the file-level metrics dataset dataset at the 100th epoch.
Regarding the balanced datasets, the LSTM model learned 93% accuracy for the class-level metrics dataset and 88% accuracy for the file-level metrics dataset dataset at the 100th epoch. The GRU model, the model learned 93% accuracy for the class-level metrics dataset and 88% accuracy for the file-level metrics dataset at the 100th epoch.

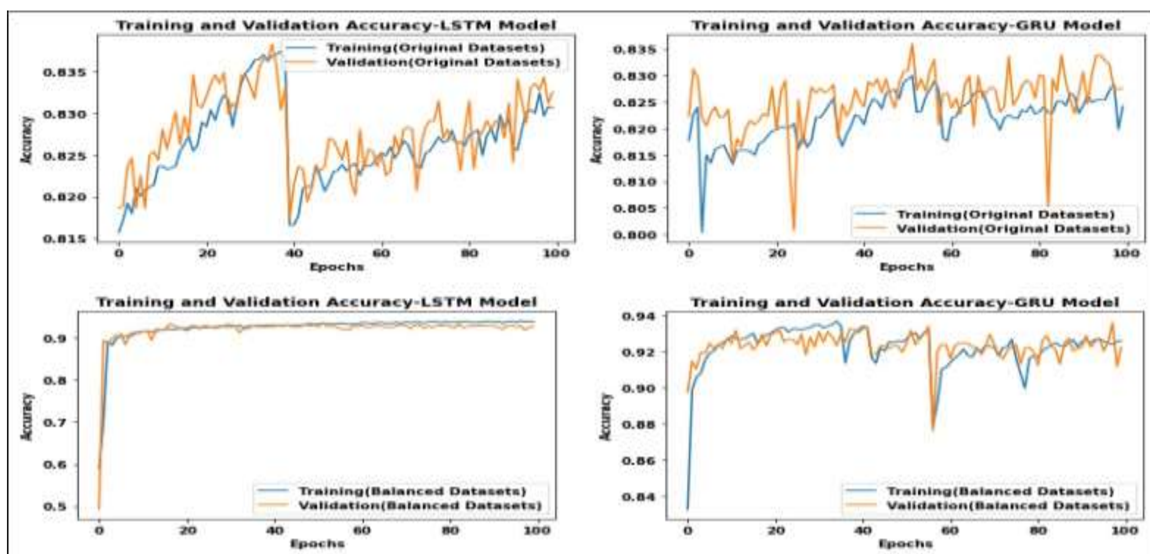


Figure 6.6 Represents the training and validation accuracy of the models across all datasets - class-level metrics

Figure 6.7 Represents the training and validation accuracy of the models across all datasets - file-level metrics

Figures 6.8 and 6.9 show the training and validation loss of the models on the original and balanced datasets. The vertical axis presents the loss of the model, and the horizontal axis illustrates the number of epochs. The loss indicates how bad a model prediction was. Regarding the original datasets, the LSTM model loss is 0.125 for the class-level metrics dataset and 0.152 for the file-level metrics dataset dataset at the 100th epoch. The GRU model loss is 0.130 for the class-level metrics dataset and 0.152 for the file-level metrics dataset at the 100th epoch. Regarding the balanced datasets, the LSTM model loss is 0.051 for the class level metrics dataset and 0.090 for the file level metrics dataset dataset at the 100th epoch. The GRU model, the model loss is 0.063 for the class-level metrics dataset and 0.093 for the file-level metrics dataset at the 100th epoch. These Figures demonstrate a consistent trend of increasing accuracy and decreasing loss as the number of epochs advances. The high accuracy achieved, and the low loss obtained serve as evidence of the effective training and validation of the proposed models.

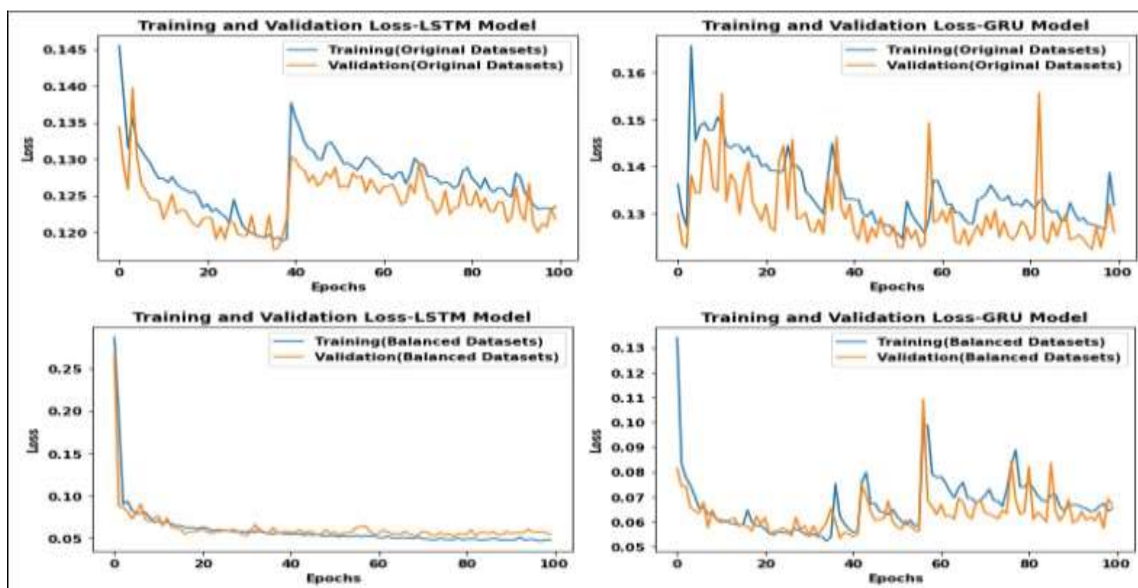


Figure 6.8 Represents the training and validation loss of the models across all datasets - class-level metrics

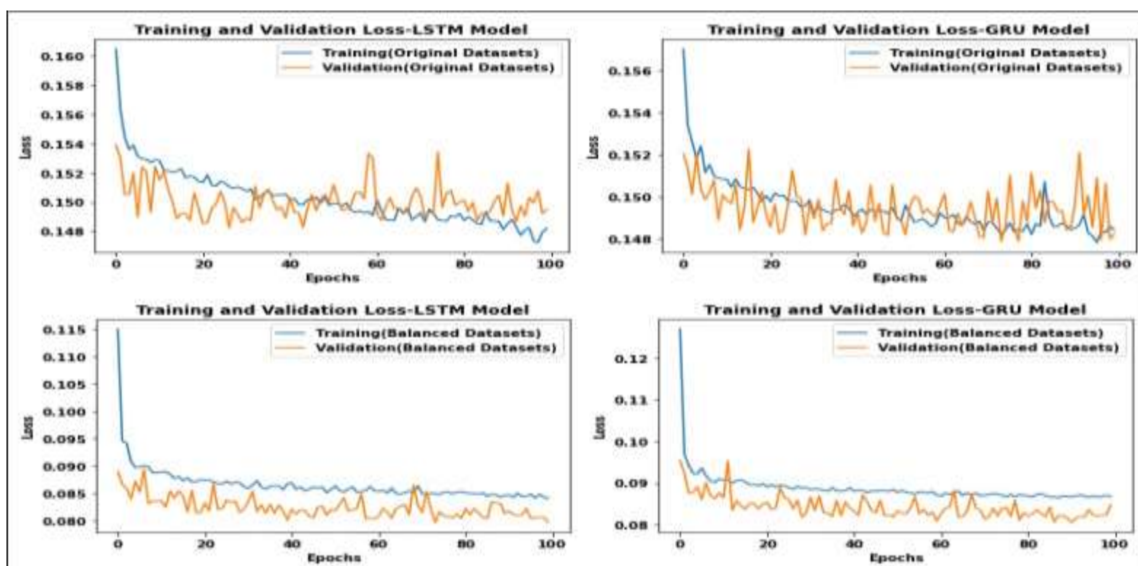


Figure 6.9 Represents the training and validation loss of the models across all datasets - file-level metrics

Figures 6.10 and 6.11 show the ROC curves of the model on the original and balanced datasets. The vertical axis presents the actual positive rate of the model, and the horizontal axis illustrates the false positive rate. The AUC is a sign of the performance of the model. The larger AUC is, the better the model performance will be. Based on the Figures, the values are encouraging and indicate our proposed models' efficiency in SBP. Regarding the original datasets, the LSTM model obtained the best AUC which is 78% on the class-level metrics data set. The worst AUC obtained by both models (LSTM and GRU) which is 75% on the file-level metrics dataset. Regarding the balanced datasets, the LSTM model obtained the best AUC which is 97% on the class-level metrics data set. The worst AUC obtained by both models (LSTM and GRU) which is 93% on the file-level metrics dataset. Further in appendix 1, Figures 1 and 2 display the AUCPR scores obtained by the proposed models on the original and balanced datasets.

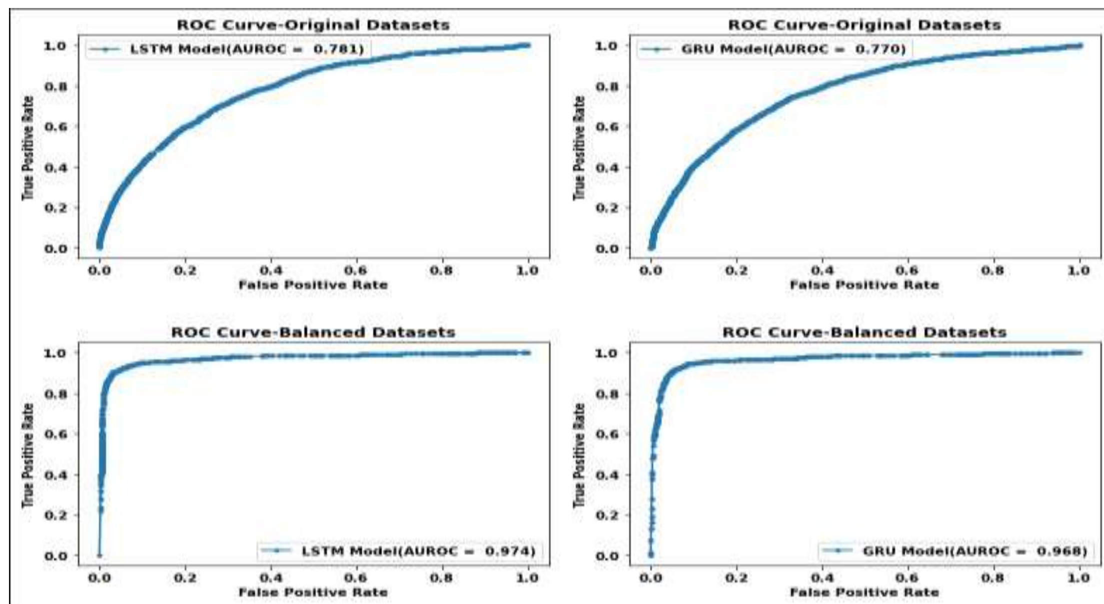


Figure 6.10 Illustrates the ROC Curves of the models across all datasets - class-level metrics

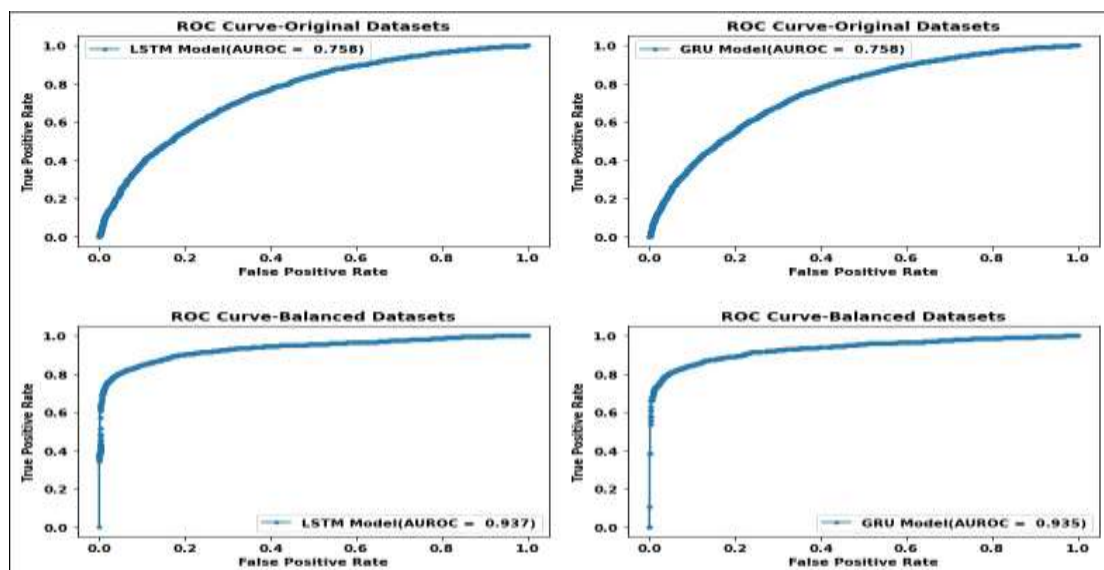


Figure 6.11 Illustrates the ROC Curves of the models across all datasets - file-level metrics

Table 6.9 shows the comparison results of our method with some previous studies based on some performance measures namely accuracy and AUC. The best values are indicated with bold text and "- "to indicate the approaches that did not provide results in a particular data set. The comprehensive findings, presented in Table 6.9, showed that while certain earlier studies displayed higher values, our proposed method surpassed other techniques on most datasets. This indicates the superior performance of our approach and its potential to outperform existing methods in the context of software bug prediction. By conducting this rigorous evaluation and providing empirical evidence, our study contributes valuable insights to the field and underscores the effectiveness of our novel approach in improving bug prediction accuracy.

Table 6.9 Comparison of the proposed approach with other existing approaches based on the accuracy and AUC

Approaches	**Datasets**	**Accuracy**	**AUC**
LSTM [2]	JIRA dataset	0.89	-
NB[21]	software fault datasets (DS1, DS2, DS3)	0.89, 0.95, 0.95	-
DT[21]	software fault datasets (DS1, DS2, DS3)	0.95, 0.97, **0.99**	-
ANNs[21]	software fault datasets (DS1, DS2, DS3)	0.93, 0.95, 0.96	-
LSTM[55]	Bug report datasets (Eclipse Platform UI, JDT)	0.67, 0.76	-
CNN and RF with Boosting[60]	Bug report datasets (Mozilla, Eclipse, JBoss, OpenFOAM, Firefox)	0.94, 0.95, 0.94, 0.98, 0.97	-
Defect prediction via attention-based RNNs (DP-ARNN)[84]	PROMISE datasets (Camel, Lucene, Poi, Xerces, Jedit, Xalan, Synapse)	-	0.79, 0.68, 0.79, 0.76, 0.82, 0.67, 0.64
Credibility-based imbalance boosting[115]	NASA datasets (CM1, KC1, PC1, JM1)	-	0.72, 0.67, 0.85, 0.67
Defect prediction through attention-based GRU-LSTM[116]	Code4Bench for C/C++ code	0.69	-
Deep Neural Networks[117]	Unified bug dataset (Bug drediction Dataset, PROMISE dataset, GitHub bug dataset)	-	0.81
Our models (LSTM, GRU)	Unified Bug Dataset_ Balanced Dataset (class-level)	0.93, 0.93	**0.97**, 0.96
Our models (LSTM, GRU)	Unified Bug Dataset_ Balanced Dataset (file-level)	0.88, 0.88	0.93, 0.93

In summary, the primary objective of this study was to present a method based on combining two RNN models namely LSTM and GRU with the Undersampling method (Near Miss) for SBP. We compared the results obtained by the proposed method based on the original and balanced datasets to investigate the impact of Undersampling methods on improving the accuracy of ML techniques. Additionally, the proposed method's results were compared with those presented in previous studies. After comparing the results obtained by the proposed models on the original datasets with results obtained by the proposed models on the balanced datasets, as shown in the Tables and Figures, we note that the models got good scores on the balanced datasets and the results improved further due to balancing, which indicated that the combination of LSTM and GRU with the Undersampling method (Near Miss) positively affects bug prediction performance in datasets with imbalanced class distributions. Moreover, data sampling methods play an essential role in improving the accuracy of the ML models in predicting software bug. Regarding evaluating the results obtained from our proposed method

and their comparison with some results of other studies, we conclude that our models are promising, competitive and outperform other models in the previous studies. Moving forward, our future work aims to evaluate the robustness of the proposed method on a wide range of datasets.

6.3 Bi-LSTM with Oversampling Methods in Software Defect Prediction (SDP)

In this sub-section, we discuss the findings of the third study, the goal was to present a method based on combining a Bi-LSTM network with Oversampling methods (Random Oversampling and SMOTE) for SDP. The experiments have been conducted based on benchmark datasets obtained from the PROMISE repository. The experimental results were evaluated and compared based on various performance measures (accuracy, precision, recall, f-measure, MCC, AUC, AUCPR, and MSE).
The performance of the prediction model is reported in Tables 6.10 to 6.15, and Figures 6.12 to 6.21, appendix 2 (Figures 1,2,3).
According to Table 6.10: Accuracy for the various original datasets: the highest accuracy was achieved by the proposed model on the *jedit* dataset, which is 97%. The lowest accuracy was achieved by the proposed model on the *ant* dataset, which is 80%. Precision for the various original datasets: the highest Precision was achieved by the proposed model on the *log4j* and *xerces* datasets, which is 95%. The proposed model achieved the lowest Precision on the *jedit* dataset, 0%. Recall for the various original datasets: the highest Recall was achieved by the proposed model on the *log4j* dataset, which is 100%. The lowest Recall was achieved by the proposed model on the *jedit* dataset, which is 0%. F-Measure for the various original datasets: the highest F-Measure was achieved by the proposed model on the *log4j* dataset, which is 97%. The lowest F-Measure was achieved by the proposed model on the *jedit* dataset, which is 0%. MCC for the various original datasets: the highest MCC was achieved by the proposed model on the *xerces* dataset, which is 75%. The lowest MCC was achieved by the proposed model on the *jedit* and *log4j* datasets, which is 0%. AUC for the various original datasets: the highest AUC was achieved by the proposed model on the *xerces* dataset, 94%. The lowest AUC was achieved by the proposed model on the *log4j* dataset, which is 60%. AUCPR for the various original datasets: the highest AUCPR was achieved by the proposed model on the *xerces* dataset, 98%. The lowest AUCPR was achieved by the proposed model on the *jedit* dataset, which is 29%. MSE for the various original datasets: the highest MSE was achieved by the proposed model on the *ant* dataset, which is 0.152. The lowest MSE was achieved by the proposed model on the *jedit* dataset, which is 0.030.

Table 6.10 Performance analysis for proposed Bi-LSTM Network - Original Datasets

Datasets	Performance Measures							
	Accuracy	**Precision**	**Recall**	**F-Measure**	**MCC**	**AUC**	**AUCPR**	**MSE**
ant	0.80	0.50	0.50	0.50	0.37	0.79	0.48	0.152
camel	0.82	0.56	0.28	0.37	0.30	0.69	0.37	0.146
ivy	0.87	0.50	0.22	0.31	0.27	0.72	0.40	0.105
jedit	0.97	0.00	0.00	0.00	0.00	0.85	0.29	0.030
log4j	0.95	0.95	1.00	0.97	0.00	0.60	0.96	0.041
xerces	0.91	0.95	0.92	0.94	0.75	0.94	0.98	0.075
Averages	**0.88**	**0.57**	**0.48**	**0.51**	**0.28**	**0.76**	**0.58**	**0.091**

According to Table 6.11: Accuracy for the various balanced datasets using Random Oversampling: the highest accuracy was achieved by the proposed model on the *jedit* and *log4j*

datasets, which is 99%. The lowest accuracy was achieved by the proposed model on the *ivy* dataset, which is 90%. Precision for the various balanced datasets using Random Oversampling: The highest Precision was achieved by the proposed model on the *log4j* dataset, which is 100%. The proposed model on the *ivy* dataset achieved the lowest Precision, which is 82%. Recall for the various balanced datasets using Random Oversampling: The highest Recall was achieved by the proposed model on the *ivy* and *jedit* datasets, which is 100%. The lowest Recall was achieved by the proposed model on the *xerces* dataset, which is 92%. F-Measure for the various balanced datasets using Random Oversampling: the highest F-Measure was achieved by the proposed model on the *jedit* and *log4j* datasets, which is 99%. The lowest F-Measure was achieved by the proposed model on the *ivy* dataset, which is 90%. MCC for the various the various balanced datasets using Random Oversampling: the highest MCC was achieved by the proposed model on the *jedit* and *log4j* datasets, which is 97%. The lowest MCC was achieved by the proposed model on the *camel* and *ivy* datasets, which is 81%. AUC for the various balanced datasets using Random Oversampling: The highest AUC was achieved by the proposed model on the *jedit* and *log4j* datasets, which is 99%. The lowest AUC was achieved by the proposed model on the *camel* and *ivy* datasets, which is 93%. AUCPR for the various balanced datasets using Random Oversampling: the highest AUCPR was achieved by the proposed model on the *jedit* and *log4j* datasets, which is 99%. The lowest AUCPR was achieved by the proposed model on the *ivy* dataset, which is 86%. MSE for the various balanced datasets using Random Oversampling: the highest MSE was achieved by the proposed model on the *ivy* dataset, which is 0.092. The lowest MSE was achieved by the proposed model on the *jedit* dataset, which is 0.009.

Table 6.11 Performance analysis for proposed Bi-LSTM Network - Balanced Datasets using Random Oversampling Technique

Datasets	**Performance Measures**							
	Accuracy	**Precision**	**Recall**	**F-Measure**	**MCC**	**AUC**	**AUCPR**	**MSE**
ant	0.91	0.89	0.94	0.91	0.82	0.95	0.93	0.073
camel	0.91	0.87	0.98	0.92	0.81	0.93	0.92	0.082
Ivy	0.90	0.82	1.00	0.90	0.81	0.93	0.86	0.092
jedit	0.99	0.98	1.00	0.99	0.97	0.99	0.99	0.009
log4j	0.99	1.00	0.98	0.99	0.97	0.99	0.99	0.012
xerces	0.95	0.98	0.92	0.95	0.89	0.97	0.98	0.049
Averages	**0.94**	**0.92**	**0.97**	**0.94**	**0.87**	**0.96**	**0.94**	**0.052**

According to Table 6.12: Accuracy for the various balanced datasets using SMOTE: the highest accuracy was achieved by the proposed model on the *log4j* dataset, which is 100%. The proposed model achieved the lowest accuracy on the ant dataset, 84%. Precision for the various balanced datasets using SMOTE: The highest Precision was achieved by the proposed model on the log4j dataset, which is 100%. The lowest Precision was achieved by the proposed model on the *ant* dataset, which is 81%. Recall for the various balanced datasets using SMOTE: the highest Recall was achieved by the proposed model on the *jedit* and *log4j* datasets, which is 100%. The lowest Recall was achieved by the proposed model on the *ant* and *camel* datasets, which is 88%. F-Measure for the various balanced datasets using SMOTE: the highest F-Measure was achieved by the proposed model on the *log4j* dataset, which is 100%. The lowest F-Measure was achieved by the proposed model on the *ant* dataset, which is 85%. MCC for the various balanced datasets using SMOTE: the highest MCC was achieved by the proposed model on the *log4j* dataset, which is 100%. The lowest MCC was achieved by the proposed model on the *ant* dataset, which is 67%. AUC for the various balanced datasets using SMOTE:

the highest AUC was achieved by the proposed model on the *log4j* dataset, which is 100%. The lowest AUC was achieved by the proposed model on the *ant* dataset, which is 90%. AUCPR for the various balanced datasets using SMOTE: the highest AUCPR was achieved by the proposed model on the *log4j* dataset, which is 100%. The lowest AUCPR was achieved by the proposed model on the *ant* and *camel* datasets, which is 91%. MSE for the various balanced datasets using SMOTE: the highest MSE was achieved by the proposed model on the *ant* dataset, which is 0.124. The lowest MSE was achieved by the proposed model on the *log4j* dataset, which is 0.001.

Table 6.12 Performance analysis for proposed Bi-LSTM Network - Balanced Datasets using SMOTE Technique

Datasets	Performance Measures							
	Accuracy	**Precision**	**Recall**	**F-Measure**	**MCC**	**AUC**	**AUCPR**	**MSE**
Ant	0.84	0.81	0.88	0.85	0.67	0.90	0.91	0.124
camel	0.87	0.89	0.88	0.89	0.74	0.91	0.91	0.113
Ivy	0.89	0.83	0.97	0.89	0.78	0.94	0.92	0.101
Jedit	0.99	0.98	1.00	0.99	0.97	0.99	0.99	0.011
log4j	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.001
xerces	0.93	0.93	0.92	0.93	0.85	0.96	0.97	0.067
Averages	**0.92**	**0.90**	**0.94**	**0.92**	**0.83**	**0.95**	**0.95**	**0.069**

Table 6.13 presents the statistical analysis results (paired t-test) of the proposed model on the original and balanced datasets (using Random Oversampling and SMOTE) in terms of mean, Standard Deviation (STD), min, max, and P value. We notice that the mean values of the Bi-LSTM model are 0.88 on the original datasets, 0.94 on the balanced datasets using Random Oversampling, and 0.92 on the balanced datasets using SMOTE. The STD values of the Bi-LSTM model are 0.06 on the original datasets, 0.04 on the balanced datasets using Random Oversampling, and 0.06 on the balanced datasets using SMOTE. The Min values of the Bi-LSTM model are 0.80 on the original datasets, 0.90 on the balanced datasets using Random Oversampling, and 0.84 on the balanced datasets using SMOTE. The Max values of the Bi-LSTM model are 0.97 on the original datasets, 0.99 on the balanced datasets using Random Oversampling, and 1.00 on the balanced datasets using SMOTE. The P value of the Bi-LSTM model is 0.01 on the original and balanced datasets using Random Oversampling and 0.00 on the original and balanced datasets using SMOTE. Based on the P value of the model on the original and balanced data sets, we note that the P value is less than 0.05, which indicates a difference between the results of the model on the original and balanced data sets.

Table 6.13 Comparison of the results of the proposed Bi-LSTM Model based on the original and balanced datasets in terms of accuracy using paired t-test

Paired t-test	**Original Datasets**	**Balanced Datasets using Random Oversampling**	**Original Datasets**	**Balanced Datasets using SMOTE**
Mean	0.88	0.94	0.88	0.92
STD	0.06	0.04	0.06	0.06
Min	0.80	0.90	0.80	0.84
Max	0.97	0.99	0.97	1.00
P value	0.01		0.00	

We used Boxplots to aggregate the achieved results to get a more accurate overview of the quality of the results. Figure 6.12 shows the Box plots for the performance measures (Accuracy, Precision, Recall, F-measure, MCC, AUC, AUCPR, and MSE) on the original and balanced datasets: The averages of (Accuracy, Precision, Recall, F-measure, MCC, AUC, AUCPR, and

MSE) on the original datasets are 0.88, 0.57, 0.48, 0.51, 0.28, 0.76, 0.58, and 0.091, respectively. The averages of (Accuracy, Precision, Recall, F-measure, MCC, AUC, AUCPR, and MSE) on the balanced data sets (using Random Oversampling) are 0.94, 0.92, 0.97, 0.94, 0.87, 0.96, 0.94, and 0.052, respectively. The averages of (Accuracy, Precision, Recall, F-measure, MCC, AUC, AUCPR, and MSE) on the balanced data sets (using SMOTE) are 0.92, 0.90, 0.94, 0.92, 0.83, 0.95, 0.95, and 0.069, respectively.

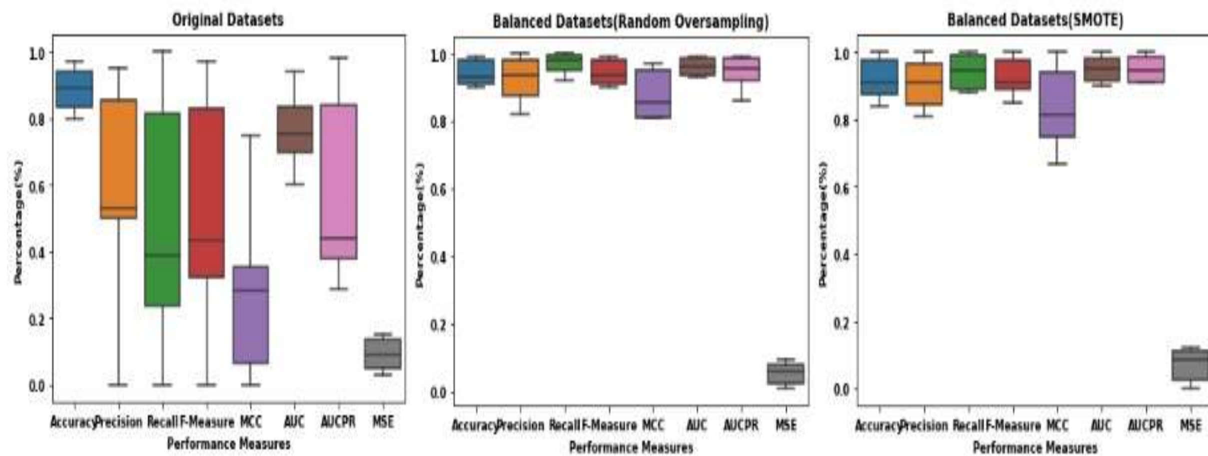


Figure 6.12 Boxplots represent performance measures obtained by the model on the original and balanced datasets

Figures 6.13 to 6.18 show the training and validation accuracy and training and validation loss of the model on the original and balanced datasets.

Figures 6.13, 6.14, and 6.15 show the training and validation accuracy of the model on the original and balanced datasets. The vertical axis presents the accuracy of the model, and the horizontal axis illustrates the number of epochs. Accuracy is the fraction of predictions that our model predicted right.

Figure 6.13 shows the accuracy values of the model on the original datasets. From the Figure, the model learned 80% accuracy for the *ant* dataset, 82% accuracy for the *camel* dataset, 87% accuracy for the *ivy* dataset, 97% accuracy for the *jedit* dataset, 95% accuracy for the log4j dataset, and 91% accuracy for *xerces* dataset at the 100th epoch.

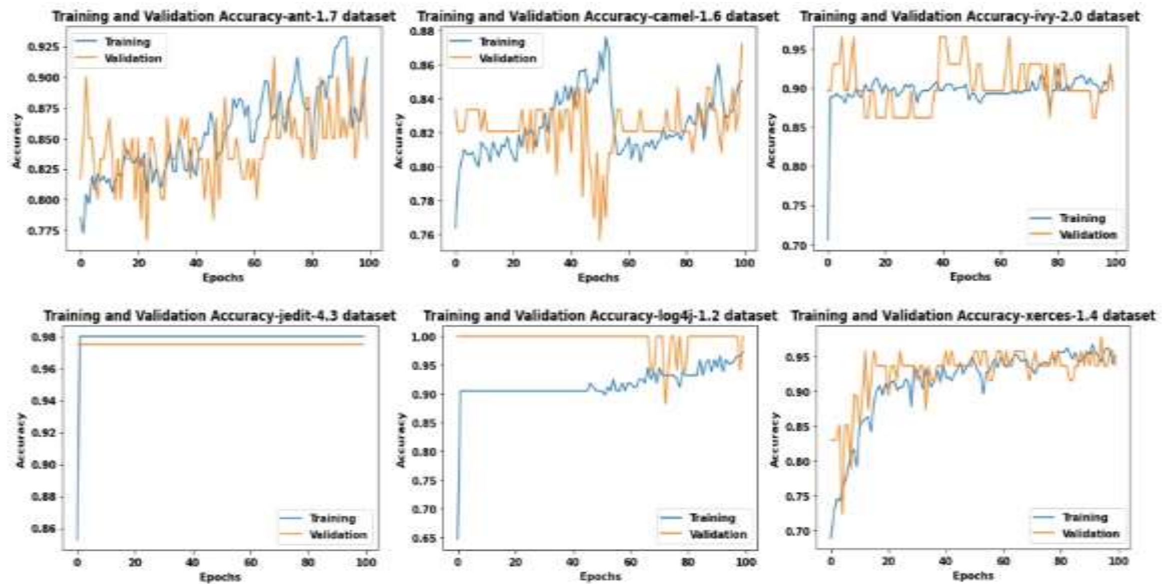


Figure 6.13 Training and validation accuracy for the original datasets

Figure 6.14 shows the accuracy values of the model on the balanced datasets (using Random Oversampling). From the Figure, the model learned 91% accuracy for the ant dataset, 91% accuracy for the camel dataset, 90% accuracy for the ivy dataset, 99% accuracy for the *jedit* dataset, 99% accuracy for the *log4j* dataset, and 95% accuracy for *xerces* dataset at the 100th epoch.

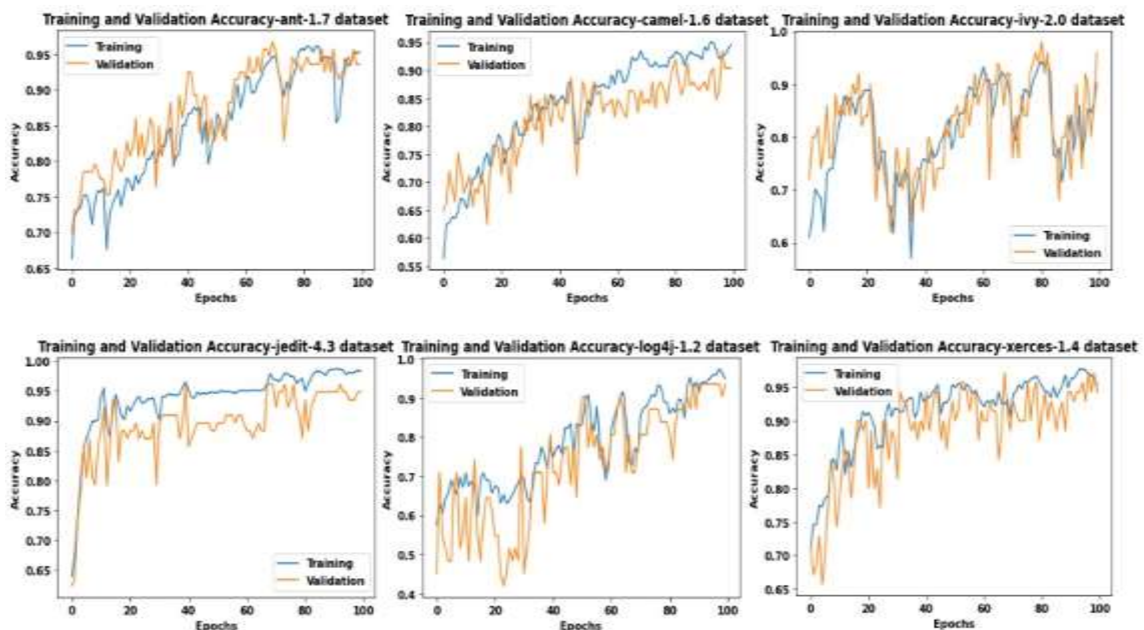


Figure 6.14 Training and validation accuracy for the balanced datasets - Random Oversampling

Figure 6.15 shows the accuracy values of the model on the balanced datasets (using SMOTE). From the Figure, the model learned 84% accuracy for the *ant* dataset, 87% accuracy for the *camel* dataset, 89% accuracy for the *ivy* dataset, 99% accuracy for the *jedit* dataset, 100% accuracy for the *log4j* dataset, and 93% accuracy for *xerces* dataset at the 100th epoch.

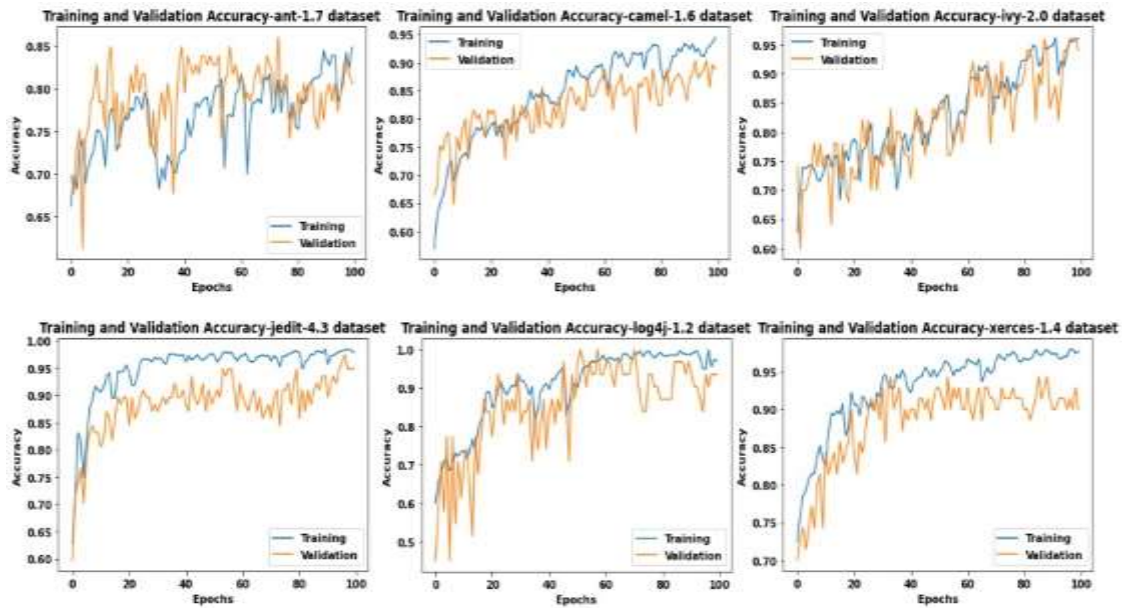


Figure 6.15 Training and validation accuracy for the balanced datasets – SMOTE

Figures 6.16, 6.17, and 6.18 show the training and validation loss of the model on the original and balanced datasets. The vertical axis presents the loss of the model, and the horizontal axis illustrates the number of epochs. The loss indicates how bad a model prediction was.

Figure 6.16 shows the loss values of the model on the original datasets. From the Figure, the model loss is 0.152 for the ant dataset, 0.146 for the *camel* dataset, 0.105 for the *ivy* dataset, 0.030 for the *jedit* dataset, 0.041 for the *log4j* dataset, and 0.075 for the *xerces* dataset at the 100th epoch.

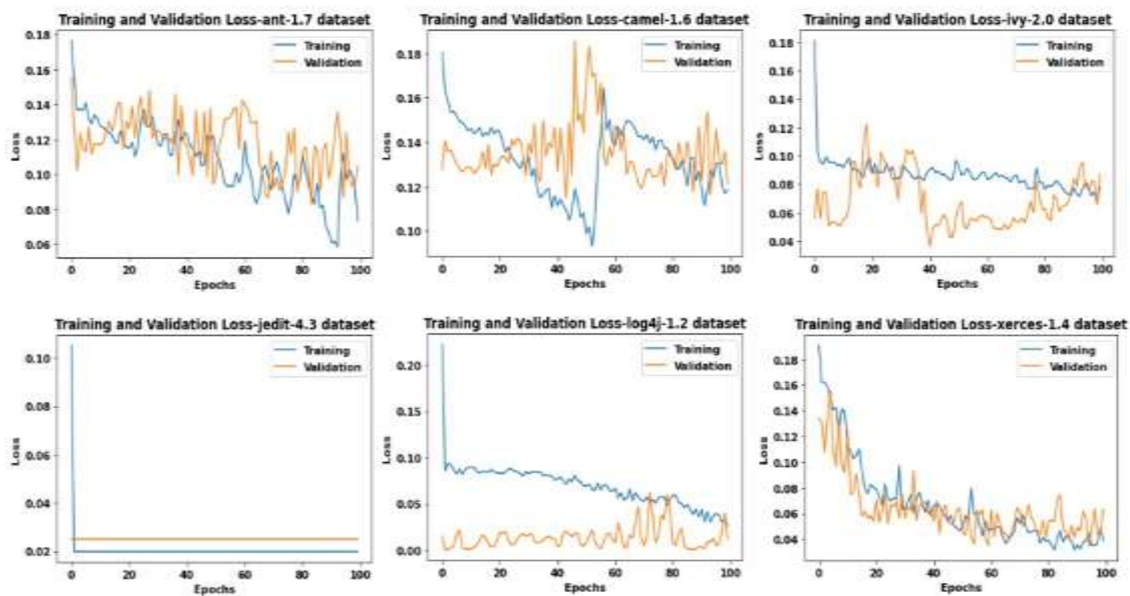


Figure 6.16 Training and validation loss for the original datasets

Figure 6.17 shows the loss values of the model on the balanced datasets (using Random Oversampling). From the Figure, the model loss is 0.073 for the ant dataset, 0.082 for the *camel* dataset, 0.092 for the *ivy* dataset, 0.009 for the *jedit* dataset, 0.012 for the *log4j* dataset, and 0.049 for the *xerces* dataset at the 100th epoch.

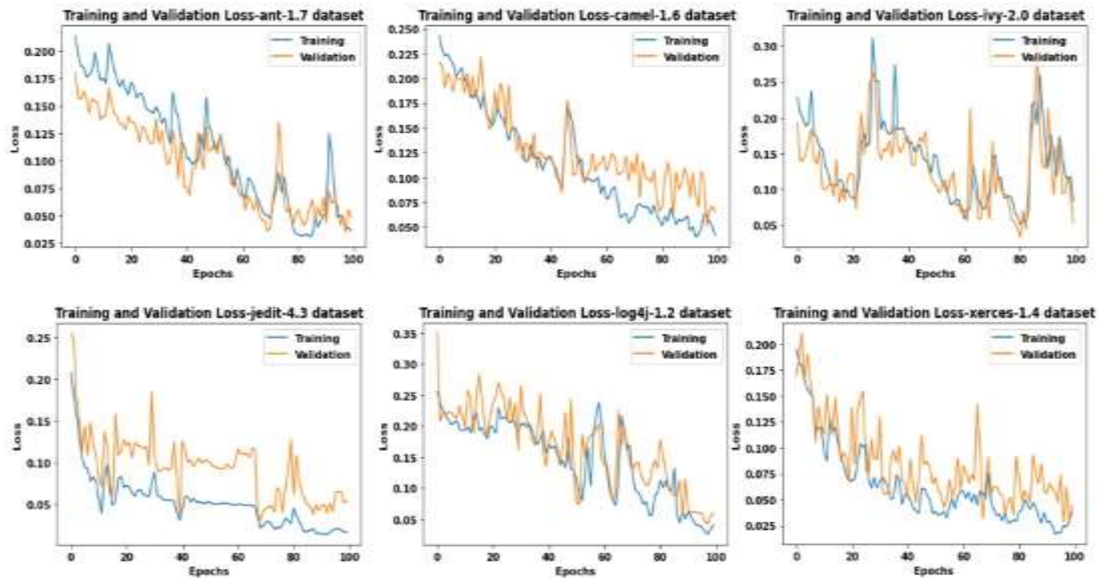


Figure 6.17 Training and validation loss for the balanced datasets - Random Oversampling

Figure 6.18 shows the loss values of the model on the balanced datasets (using SMOTE). From the Figure, the model loss is 0.124 for the *ant* dataset, 0.113 for the *camel* dataset, 0.101 for the *ivy* dataset, 0.011 for the *jedit* dataset, 0.001 for the *log4j* dataset, and 0.067 for the *xerces* dataset at the 100th epoch. As shown in the Figures, the accuracy of training and validation increases, and the loss decreases with increasing epochs. Regarding the high accuracy and low loss obtained by the proposed model, we note that the model is well-trained and validated.

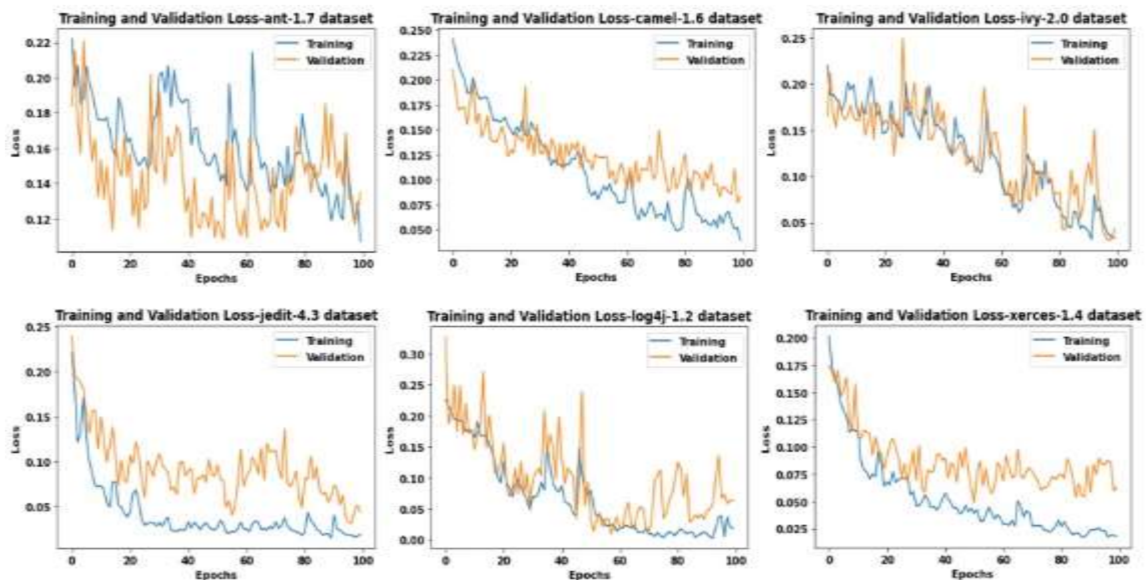


Figure 6.18 Training and validation loss for the balanced datasets - SMOTE

Figures 6.19 to 6.21 show the ROC curves of the model on the original and balanced datasets. The vertical axis presents the actual positive rate of the model, and the horizontal axis illustrates the false positive rate. The AUC is a sign of the performance of the model. The larger AUC is, the better the model performance will be. Based on the Figures, the values are encouraging and indicate our proposed model efficiency in SDP. The best AUC obtained by the proposed model in the original data sets is 94% on the *xerces* data set. The worst AUC is 60% on the *log4j* data

set. The best AUC obtained by the proposed model in the balanced data sets (using Random Oversampling) is 99% on the *jedit* and *log4j* data sets, while the worst AUC is 93% on the *camel* and *ivy* data sets. The best AUC obtained by the proposed model in the balanced data sets (using SMOTE) is 100% on the *log4j* data set, while the worst AUC is 90% on the *ant* data set. Further in appendix 2, Figures 1 to 3 show the AUCPR of the model on the original and balanced datasets.

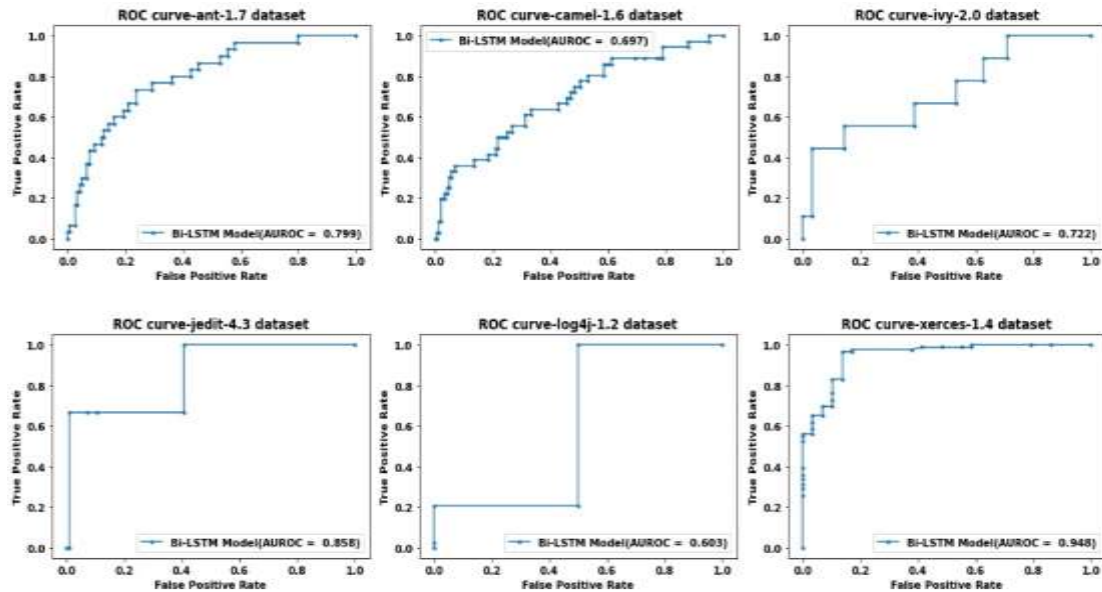


Figure 6.19 ROC curves for the original datasets

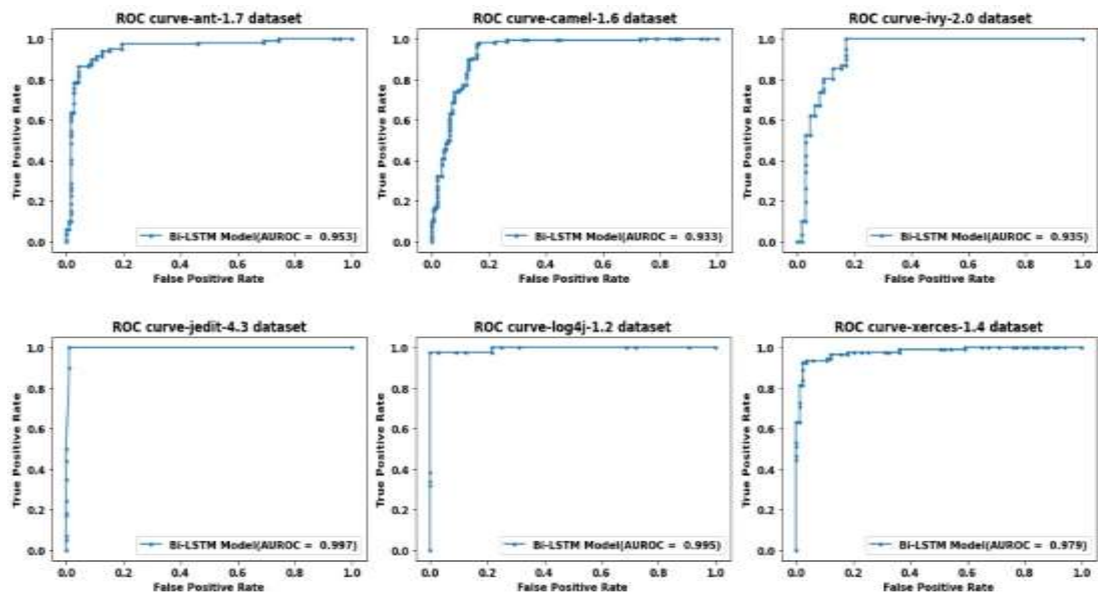


Figure 6.20 ROC curves for the balanced datasets- Random Oversampling

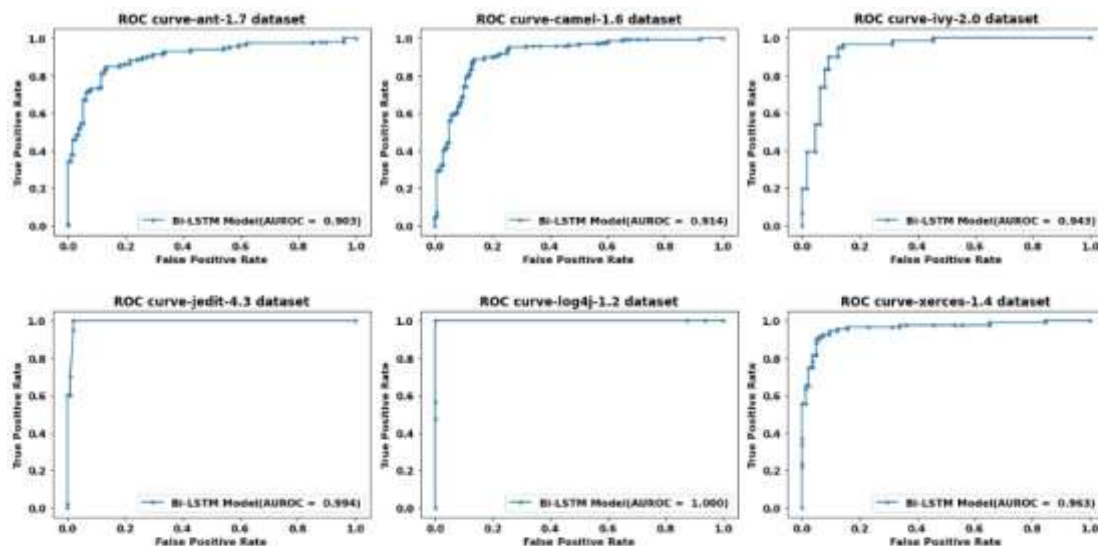


Figure 6.21 ROC curves for the balanced datasets- SMOTE

Table 6.14 shows the comparison of the results produced using our models with those obtained using the baseline model (RF) based on six performance measures: accuracy precision, recall, f-Measure, MCC, and AUC. We also compared the results produced using our model with those obtained in previous studies based on six performance measures: accuracy precision, recall, f-measure, MCC, and AUC. Table 6.15 compares the values of performance measures obtained by our Bi-LSTM network and the performance values in previous studies. The best values are indicated with bold text and "- " indicate the approaches that did not provide results in a particular data set. According to Table 6.15, some of the results in the previous studies are better than ours. Still, in most cases, our model outperforms the other state-of-the-art approaches and provides better predictive performance.

Table 6.14 Performance measures of the baseline model (RF) and Bi-LSTM

Models	**Datasets**	**Performance Measures**					
		Accuracy	**Precision**	**Recall**	**F-Measure**	**MCC**	**AUC**
RF	ant	0.81	0.53	0.53	0.53	0.41	0.70
	camel	0.81	0.47	0.19	0.27	0.20	0.57
	ivy	0.89	0.57	0.44	0.50	0.44	0.69
	jedit	0.97	0.00	0.00	0.00	0.00	0.50
	log4j	0.98	0.97	1.00	0.99	0.69	0.75
	xerces	0.95	0.96	0.98	0.97	0.86	0.92
Averages		**0.90**	**0.58**	**0.52**	**0.54**	**0.43**	**0.58**
Bi-LSTM with Random Oversampling Technique	ant	0.91	0.89	0.94	0.91	0.82	0.95
	camel	0.91	0.87	0.98	0.92	0.81	0.93
	ivy	0.90	0.82	1.00	0.90	0.81	0.93
	jedit	0.99	0.98	1.00	0.99	0.97	0.99
	log4j	0.99	1.00	0.98	0.99	0.97	0.99
	xerces	0.95	0.98	0.92	0.95	0.89	0.97
Averages		**0.94**	**0.92**	**0.97**	**0.94**	**0.87**	**0.96**
Bi-LSTM with SMOTE Technique	ant	0.84	0.81	0.88	0.85	0.67	0.90
	camel	0.87	0.89	0.88	0.89	0.74	0.91
	ivy	0.89	0.83	0.97	0.89	0.78	0.94
	jedit	0.99	0.98	1.00	0.99	0.97	0.99
	log4j	1.00	1.00	1.00	1.00	1.00	1.00
	xerces	0.93	0.93	0.92	0.93	0.85	0.96
Averages		**0.92**	**0.90**	**0.94**	**0.92**	**0.83**	**0.95**

Table 6.15 Comparison of the proposed Bi-LSTM with other existing approaches

Approaches	Datasets	Performance Measures					
		Accuracy	Precision	Recall	F-Measure	MCC	AUC
CNN[7]	ant, camel, ivy, jedit, log4j, xerces	0.85, 0.84, 0.95, 0.97, 0.97, 0.95	0.87, 0.81, 0.92, 0.94, 0.98, 0.93	0.82, 0.90, 0.98, **1.00**, 0.98, 0.98	0.85, 0.85, 0.95, 0.97, 0.98, 0.95	0.69, 0.69, 0.90, 0.93, 0.94, 0.90	0.91, 0.90, 0.98, 0.96, 0.99, 0.98
GRU[7]	ant, camel, ivy, jedit, log4j, xerces	0.83, 0.82, 0.95, 0.99, 0.96, 0.93	0.88, 0.82, 0.95, 0.98, 0.98, 0.92	0.81, 0.82, 0.95, **1.00**, 0.95, 0.94	0.85, 0.82, 0.95, 0.99, 0.96, 0.93	0.67, 0.63, 0.90, 0.97, 0.91, 0.85	0.89, 0.87, 0.98, **1.00**, 0.98, 0.97
LSTM[40]	Unified bug dataset (class-level, file-level)	0.93, 0.88	0.95, 0.94	0.92, 0.81	0.93, 0.87	0.86, 0.76	0.97, 0.93
GRU[40]	unified bug dataset (class-level, file-level)	0.93, 0.88	0.94, 0.94	0.92, 0.81	0.93, 0.87	0.86, 0.76	0.96, 0.93
Hybrid Neural Network model[46]	JEdit, IVY, Ant, Camel	0.97, 0.88, 0.81, 0.81	**1.00**, 0.99, 0.93, **1.00**	**1.00**, 0.88, 0.84, 0.81	0.98, 0.93, 0.88, 0.89	-	-
LSTM[48]	Camel		0.51	0.41	0.46	-	-
LSTM[55]	Bug report datasets (Eclipse platform UI and JDT)	0.67, 0.76	0.70, 0.76	0.86, **1.00**	0.77, 0.86	-	-
CNN[79]	ant, camel, ivy, jedit, log4j	-	-	-	0.39, 0.52, 0.31, 0.00, 0.97	0.30, 0.42, 0.25, 0.00, 0.00	-
BPDET[81]	CM1, JM1, KC1, MC1, PC1, MW1	-	-	-	0.84, 0.76, 0.83, 0.96, 0.92, 0.90	0.42, 0.23, 0.33, 0.14, 0.38, 0.33	0.75, 0.75, 0.81, 0.85, 0.88, 0.77
DP-ARNN[84]	Camel, Xerces, JEdit	-	-	-	0.51, 0.27, 0.56	-	0.79, 0.76, 0.82
LR[96]	Ant, Camel, IVY	-	-	-	0.52, 0.34, 0.30	-	-
K-NN[96]	Ant, Camel, IVY	-	-	-	0.53, 0.37, 0.30	-	-
MLP[96]	Ant, Camel, IVY	-	-	-	0.50, 0.38, 0.25	-	-
SVM[96]	Ant, Camel, IVY	-	-	-	0.50, 0.084, 0.28	-	-
CBIL[103]	Camel, JEdit, Xerces	-	-	-	0.93, 0.85, 0.95	-	0.96, 0.91, 0.98
LSTM[104]	Camel, Jedit, Log4j, Xerces	-	-	-	0.37, 0.44, 0.52, 0.26	-	-
HyGRAR[106]	JEdit, Ant	0.98, 0.96	0.70, 0.98	0.63, 0.85	-		0.81, 0.92
SPFCNN[107]	CM1, JM1, KC1, PC1, MW1	-	-	-	-	0.85, 0.74, 0.78, 0.87, 0.80	0.92, 0.87, 0.88, 0.93, 0.90
Our model (Bi-LSTM with Random Oversampling Technique)	ant, camel, ivy, jedit, log4j, xerces	0.91, 0.91, 0.90, 0.99, 0.99, 0.95	0.89, 0.87, 0.82, 0.98, **1.00**, 0.98	0.94, 0.98, **1.00**, **1.00**, 0.98, 0.92	0.91, 0.92, 0.90, 0.99, 0.99, 0.95	0.82, 0.81, 0.81, 0.97, 0.97, 0.89	0.95, 0.93, 0.93, 0.99, 0.99, 0.97
Our model (Bi-LSTM with SMOTE Technique)	ant, camel, ivy, jedit, log4j, xerces	0.84, 0.87, 0.89, 0.99, **1.00**, 0.93	0.81, 0.89, 0.83, 0.98, **1.00**, 0.93	0.88, 0.88, 0.97, **1.00**, **1.00**, 0.92	0.85, 0.89, 0.89, 0.99, **1.00**, 0.93	0.67, 0.74, 0.78, 0.97, **1.00**, 0.85	0.90, 0.91, 0.94, 0.99, **1.00**, 0.96

In summary, this study aimed to present a method based on combining a Bi-LSTM network with Oversampling methods (Random Oversampling and SMOTE) for SDP. We compared the results obtained by the proposed method based on the original and balanced datasets to investigate the impact of Oversampling methods on improving the accuracy of ML techniques. Additionally, the proposed method's results were compared with those presented in previous studies. After comparing the results obtained by the proposed model on the original datasets with results obtained by the proposed model on the balanced datasets, as shown in the Tables and Figures, we note that the model got good scores on the balanced datasets and the results improved further due to balancing, which indicated that the combination of a Bi-LSTM network with Oversampling methods (Random Oversampling and SMOTE) positively affects defect prediction performance in datasets with imbalanced class distributions. Moreover, data sampling methods play an essential role in improving the accuracy of ML models in SDP. Regarding evaluating the results obtained from our proposed method and their comparison with some results of other studies, we conclude that our model is promising in predicting software defects and outperforms other models in the previous studies. Additionally, this research has significant implications for software developers and practitioners who aim to improve software quality and reduce the risk of defects in software systems.

6.4 CNN and GRU with Hybrid (combined)-Sampling Methods in SDP

In this sub-section, we discuss the findings of the fourth study. The goal was to propose a novel SDP approach based on CNN and GRU combined with hybrid sampling method (SMOTE Tomek) for SDP. The experiments were conducted based on benchmark datasets from the PROMISE repository. The experimental results were evaluated and compared based on various performance measures (accuracy, precision, recall, f-measure, MCC, AUC, AUCPR, and MSE).

The performance of the prediction models is reported in Tables 6.16 to 6.25, and Figures 6.22 to 6.34, appendix 3 (Figures 1 to 4).

Table 6.16 Performance analysis for proposed CNN Model-Original Data sets

Datasets	Performance Measures							
	Accuracy	**Precision**	**Recall**	**F-Measure**	**MCC**	**AUC**	**AUCPR**	**MSE**
ant	0.83	0.67	0.33	0.44	0.38	0.82	0.57	0.131
camel	0.82	0.62	0.14	0.23	0.23	0.74	0.39	0.136
ivy	0.90	0.67	0.44	0.53	0.49	0.81	0.53	0.086
jedit	0.96	0.00	0.00	0.00	0.01	0.83	0.07	0.037
log4j	0.95	0.95	1.00	0.97	0.00	0.46	0.93	0.048
xerces	0.94	0.94	0.99	0.96	0.83	0.95	0.98	0.049
Averages	**0.90**	**0.64**	**0.48**	**0.52**	**0.32**	**0.76**	**0.57**	**0.081**

Table 6.17 Performance analysis for proposed CNN Model-Balanced Datasets

Datasets	Performance Measures							
	Accuracy	**Precision**	**Recall**	**F-Measure**	**MCC**	**AUC**	**AUCPR**	**MSE**
ant	0.85	0.87	0.82	0.85	0.69	0.91	0.92	0.117
camel	0.84	0.81	0.90	0.85	0.69	0.90	0.89	0.132
ivy	0.95	0.92	0.98	0.95	0.90	0.98	0.96	0.051
jedit	0.97	0.94	1.00	0.97	0.93	0.96	0.88	0.027
log4j	0.97	0.98	0.98	0.98	0.94	0.99	0.99	0.028
xerces	0.95	0.93	0.98	0.95	0.90	0.98	0.98	0.043
Averages	**0.92**	**0.90**	**0.94**	**0.92**	**0.84**	**0.95**	**0.93**	**0.066**

Table 6.18 Performance analysis for proposed GRU Model-Original Data sets

Datasets	Performance Measures							
	Accuracy	Precision	Recall	F-Measure	MCC	AUC	AUCPR	MSE
ant	0.81	0.52	0.47	0.49	0.37	0.73	0.47	0.152
camel	0.79	0.30	0.08	0.13	0.06	0.70	0.31	0.146
ivy	0.92	0.80	0.44	0.57	0.55	0.71	0.56	0.076
jedit	0.97	0.00	0.00	0.00	0.00	0.93	0.24	0.028
log4j	0.95	0.95	1.00	0.97	0.00	0.29	0.93	0.048
xerces	0.91	0.92	0.96	0.94	0.74	0.89	0.91	0.090
Averages	**0.89**	**0.58**	**0.49**	**0.51**	**0.28**	**0.70**	**0.57**	**0.090**

Table 6.19 Performance analysis for proposed GRU Model-Balanced Datasets

Datasets	Performance Measures							
	Accuracy	Precision	Recall	F-Measure	MCC	AUC	AUCPR	MSE
ant	0.83	0.88	0.81	0.85	0.67	0.89	0.89	0.130
camel	0.82	0.82	0.82	0.82	0.63	0.87	0.84	0.144
ivy	0.95	0.95	0.95	0.95	0.90	0.98	0.99	0.055
jedit	0.99	0.98	1.00	0.99	0.97	1.00	1.00	0.026
log4j	0.96	0.98	0.95	0.96	0.91	0.98	0.98	0.073
xerces	0.93	0.92	0.94	0.93	0.85	0.97	0.98	0.064
Averages	**0.91**	**0.92**	**0.91**	**0.91**	**0.82**	**0.94**	**0.94**	**0.082**

Table 6.20 Performance analysis for proposed models based on precision and recall measures - CNN Model

Original Data sets	Performance Measures			
	Precision		Recall	
	Defective class	Non-defective class	Defective class	Non-defective class
ant	0.67	0.85	0.33	0.96
camel	0.62	0.83	0.14	0.98
ivy	0.67	0.92	0.44	0.97
jedit	0.00	0.97	0.00	0.99
log4j	0.95	0.00	1.00	0.00
xerces	0.94	0.96	0.99	0.79
Averages	**0.64**	**0.75**	**0.48**	**0.78**
Balanced Datasets	**Performance Measures**			
	Precision		**Recall**	
	Defective class	**Non-defective class**	**Defective class**	**Non-defective class**
ant	0.87	0.82	0.82	0.87
camel	0.81	0.89	0.90	0.79
ivy	0.92	0.98	0.98	0.91
jedit	0.94	1.00	1.00	0.94
log4j	0.98	0.97	0.98	0.97
xerces	0.93	0.98	0.98	0.93
Averages	**0.90**	**0.94**	**0.94**	**0.90**

Table 6.21 Performance analysis for proposed models based on precision and recall measures - GRU Model

Original Data sets	Performance Measures			
	Precision		Recall	
	Defective class	Non-defective class	Defective class	Non-defective class
ant	0.52	0.87	0.47	0.89
camel	0.30	0.82	0.08	0.96
ivy	0.80	0.92	0.44	0.98
jedit	0.00	0.97	0.00	1.00
log4j	0.95	0.00	1.00	0.00

xerces	0.92	0.85	0.96	0.76
Averages	**0.58**	**0.73**	**0.49**	**0.76**
Balanced Datasets	**Performance Measures**			
	Precision		**Recall**	
	Defective class	**Non-defective class**	**Defective class**	**Non-defective class**
ant	0.88	0.79	0.81	0.86
camel	0.82	0.82	0.82	0.82
ivy	0.95	0.95	0.95	0.95
jedit	0.98	1.00	1.00	0.98
log4j	0.98	0.94	0.95	0.97
xerces	0.92	0.94	0.94	0.91
Averages	**0.92**	**0.90**	**0.91**	**0.91**

Table 6.22 Summarizes the range of measures values for the proposed models on the original and balanced datasets

Model	Accuracy	Precision	Recall	F-measure	MCC	AUC	AUCPR	MSE
CNN model on the original datasets	0.82 to 0.96	0.00 to 0.95	0.00 to 1.00	0.00 to 0.97	0.00 to 0.83	0.46 to 0.95	0.07 to 0.98	0.037 to 0.136
CNN model on the balanced datasets	0.84 to 0.97	0.81 to 0.98	0.82 to 1.00	0.85 to 0.98	0.69 to 0.94	0.90 to 0.99	0.88 to 0.99	0.027 to 0.132
GRU model on the original datasets	0.79 to 0.97	0.00 to 0.95	0.00 to 1.00	0.00 to 0.97	0.00 to 0.74	0.29 to 0.93	0.24 to 0.93	00.028 to 0.152
GRU model on the balanced datasets	0.82 to 0.99	0.82 to 0.98	0.81 to 1.00	0.82 to 0.99	0.63 to 0.97	0.87 to 1.00	0.84 to 1.00	0.026 to 0.144

Table 6.23 presents the statistical analysis results (paired t-test) of proposed models on the original and balanced datasets regarding mean, Standard Deviation (STD), min, max, and P value. We notice that the mean values of the CNN model are 0.90 on the original datasets and 0.92 on the balanced datasets, while the mean values of the GRU model are 0.89 on the original datasets and 0.91 on the balanced datasets. The STD values of the CNN model are 0.06 on the original datasets and 0.06 on the balanced datasets, while the STD values of the GRU model are 0.07 on the original datasets and 0.07 on the balanced datasets. The Min values of the CNN model are 0.82 on the original datasets and 0.84 on the balanced datasets, while the Min values of the GRU model are 0.79 on the original datasets and 0.82 on the balanced datasets. The Max values of the CNN model are 0.96 on the original datasets and 0.97 on the balanced datasets, while the Max values of the GRU model are 0.97 on the original datasets and 0.99 on the balanced datasets. The P value of the CNN model is 0.015 based on the original and balanced datasets, while the P value of the GRU model is 0.000 based on the original and balanced datasets. Based on the P value of both models on the original and balanced data sets, we note that the P value is less than 0.05, indicating a difference between the results of the models on the original and balanced data sets.

Table 6.23 Comparison of the proposed models in terms of accuracy using paired t-test

Paired t-test	CNN Model		GRU Model	
	Original Datasets	Balanced Datasets	Original Datasets	Balanced Datasets
Mean	0.90	0.92	0.89	0.91
STD	0.06	0.06	0.07	0.07
Min	0.82	0.84	0.79	0.82
Max	0.96	0.97	0.97	0.99
P value	0.015		0.000	

We used Boxplots to aggregate the achieved results to get a more accurate overview of the quality of the results. Figure 6.22 shows the Box plots of performance measures for the original and balanced datasets (Accuracy, Precision, Recall, F-measure, MCC, AUC, AUCPR, and MSE). The CNN model averages on the original datasets (Accuracy, Precision, Recall, F-measure, MCC, AUC, AUCPR, and MSE) are 0.90, 0.64, 0.48, 0.52, 0.32, 0.76, 0.57, and 0.081, respectively. The CNN model averages on the balanced data sets (Accuracy, Precision, Recall, F-measure, MCC, AUC, AUCPR, and MSE) are 0.92, 0.90, 0.94, 0.92, 0.84, 0.95, 0.93, and 0.066, respectively. The GRU model averages on the original datasets (Accuracy, Precision, Recall, F-measure, MCC, AUC, AUCPR, and MSE) are 0.89, 0.58, 0.49, 0.51, 0.28, 0.70, 0.57, and 0.090, respectively. The averages of (Accuracy, Precision, Recall, F-measure, MCC, AUC, AUCPR, and MSE) of the GRU model on the balanced data sets are 0.91, 0.92, 0.91, 0.91, 0.82, 0.94, 0.94, and 0.082, respectively.

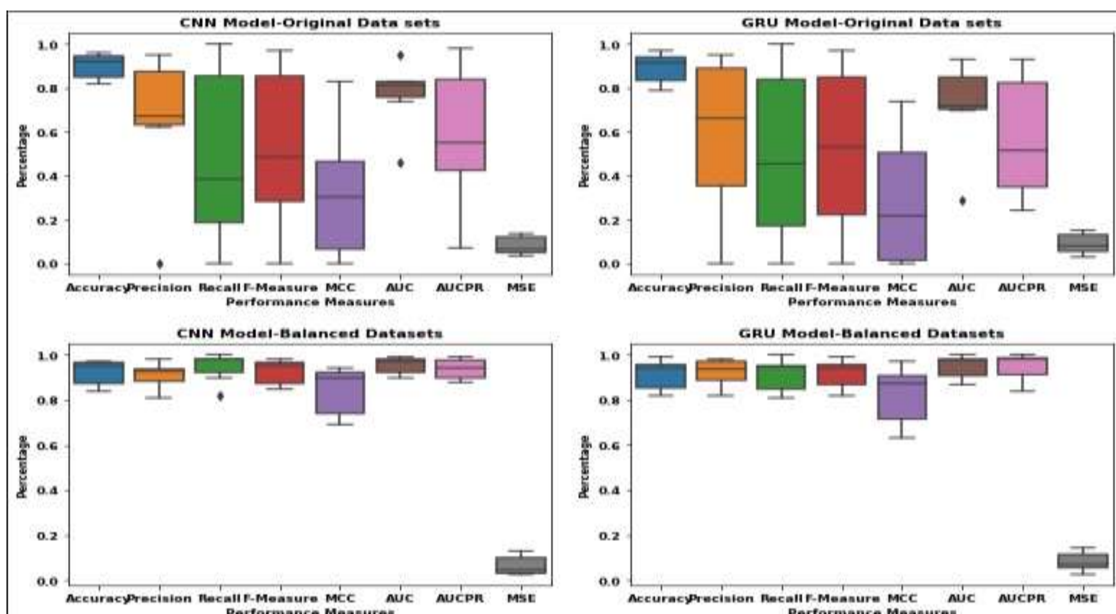


Figure 6.22 Boxplots represent performance measures obtained by proposed models on all datasets

Figures 6.23 to 6.30 show the training and validation accuracy and training and validation loss of the models on the original and balanced datasets.

Figures 6.23 to 6.26 show the training and validation accuracy of the models. The vertical axis presents the accuracy of the model, and the horizontal axis illustrates the number of epochs. Accuracy is the fraction of predictions that our model predicted right.

Figure 6.23 shows the accuracy values of the CNN model on the original data sets. The accuracy values are 0.83 on the *ant* data set, 0.82 on the *camel* data set, 0.90 on the *ivy* data set, 0.96 on the *jedit* data set, 0.95 on the *log4j* data set, and 0.94 on the *xerces* data set.

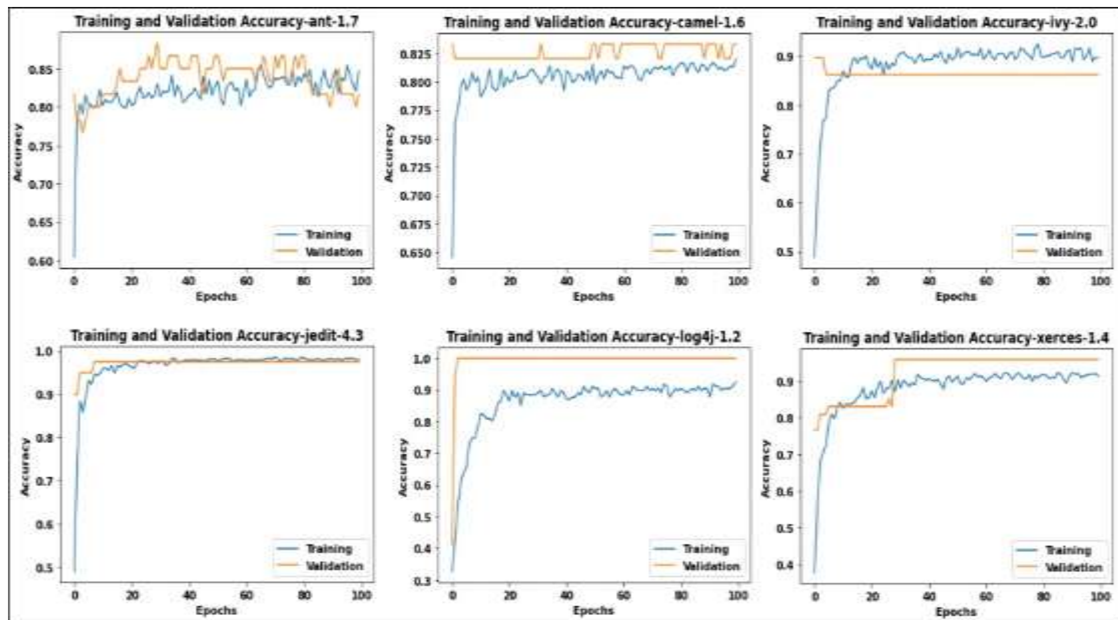


Figure 6.23 Training and Validation Accuracy for the original data sets - CNN model

Figure 6.24 shows the accuracy values of the CNN model on the balanced data sets. The accuracy values are 0.85 on the *ant* data set, 0.84 on the *camel* data set, 0.95 on the *ivy* data set, 0.97 on the *jedit* data set, 0.97 on the *log4j* data set, and 0.95 on the *xerces* data set.

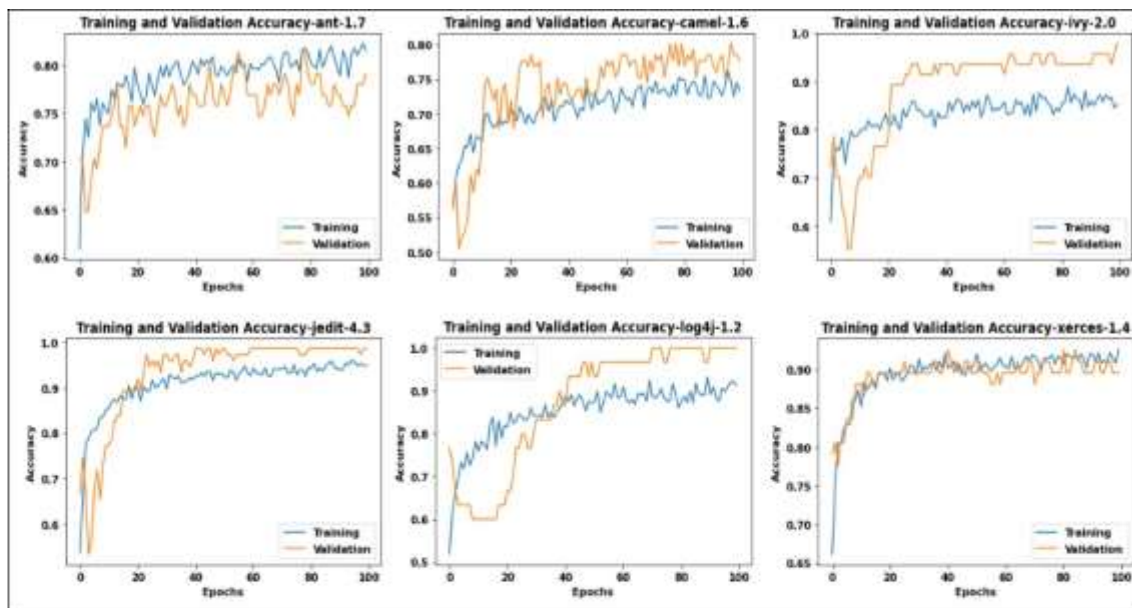


Figure 6.24 Training and Validation Accuracy for the balanced data sets - CNN model

Figure 6.25 shows the accuracy values of the GRU model on the original data sets. The accuracy values are 0.81 on the *ant* data set, 0.79 on the *camel* data set, 0.92 on the ivy data set, 0.97 on the *jedit* data set, 0.95 on the *log4j* data set, and 0.91 on the *xerces* data set.

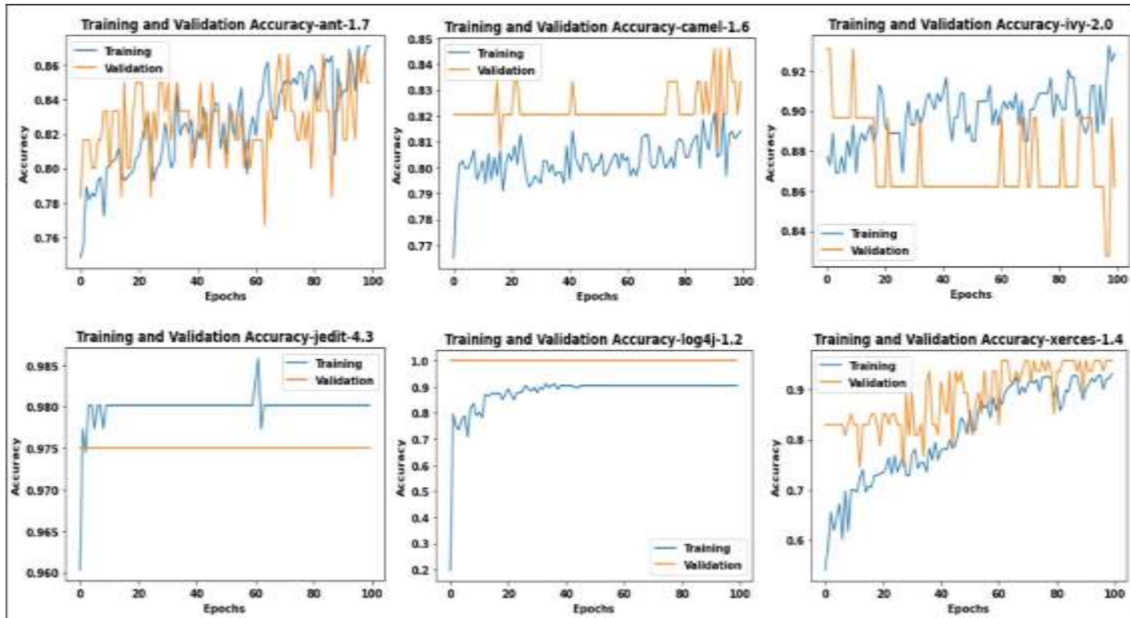


Figure 6.25 Training and Validation Accuracy for the original data sets - GRU model

Figure 6.26 shows the accuracy values of the GRU model on the balanced datasets. The accuracy values are 0.83 on the ant data set, 0.82 on the *camel* data set, 0.95 on the *ivy* data set, 0.99 on the *jedit* data set, 0.96 on the *log4j* data set, and 0.93 on the *xerces* data set.

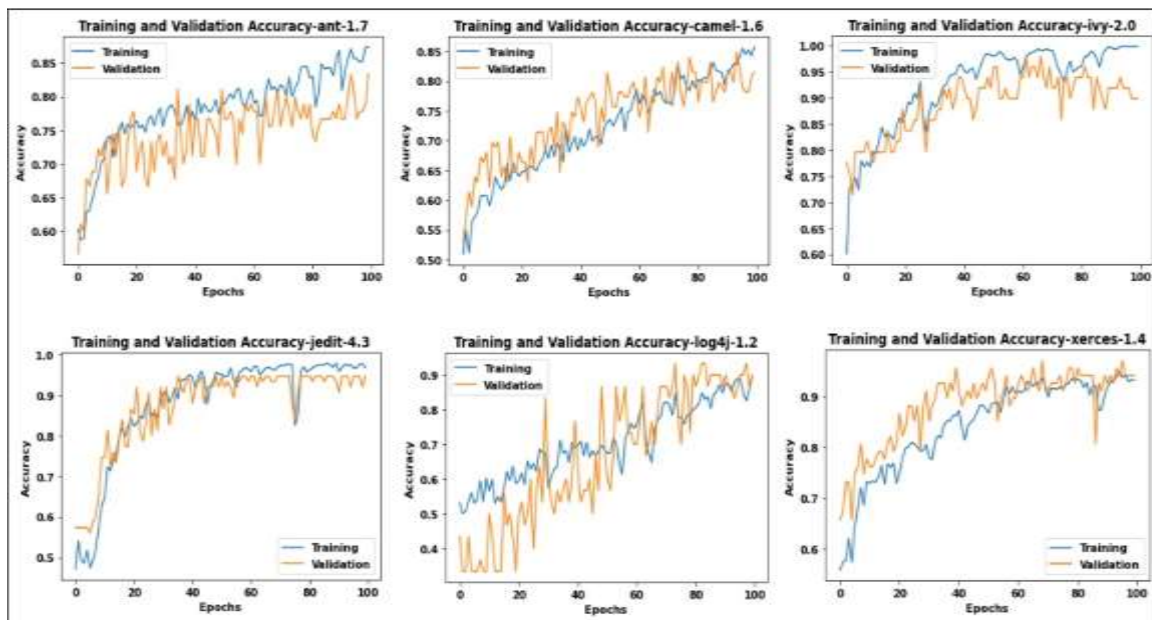


Figure 6.26 Training and Validation Accuracy for the balanced data sets - GRU model

Figures 6.27 to 6.30 show the training and validation loss of the models. The vertical axis presents the loss of the model, and the horizontal axis illustrates the number of epochs. The loss indicates how bad a model prediction was.
Figure 6.27 shows the loss values of the CNN model on the original data sets. The loss values are 0.131 on the *ant* data set, 0.136 on the *camel* data set, 0.086 on the *ivy* data set, 0.037 on the *jedit* data set, 0.048 on the *log4j* data set, and 0.049 on the *xerces* data set.

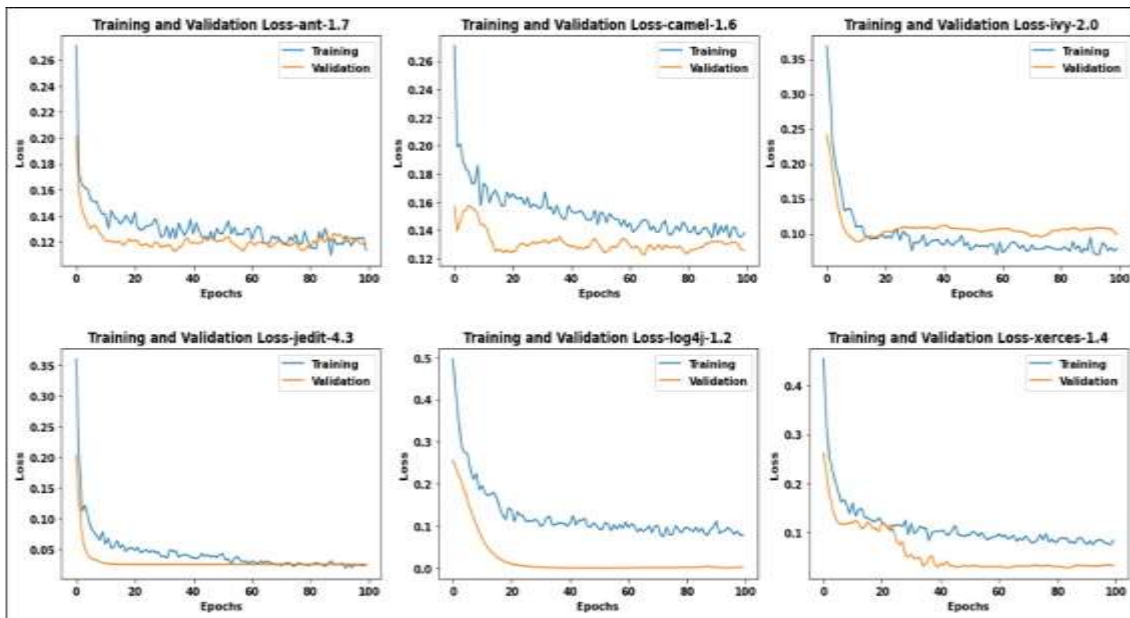


Figure 6.27 Training and Validation Loss for the original data sets - CNN model

Figure 6.28 shows the loss values of the CNN model on the balanced data sets. The loss values are 0.117 on the *ant* data set, 0.132 on the *camel* data set, 0.051 on the *ivy* data set, 0.027 on the *jedit* data set, 0.028 on the *log4j* data set, and 0.043 on the *xerces* data set.

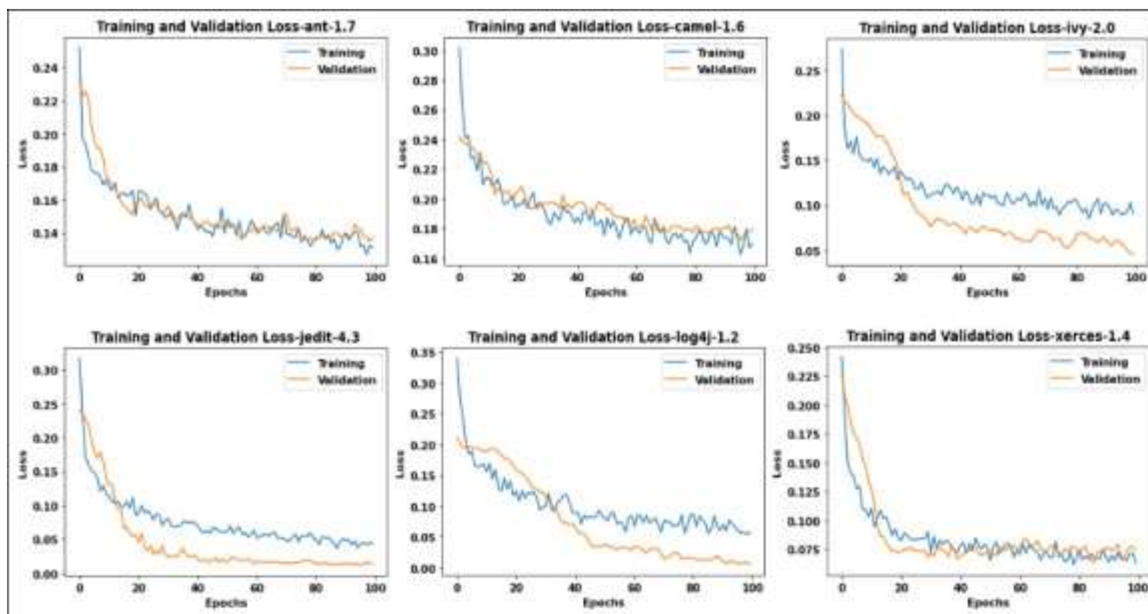


Figure 6.28 Training and Validation Loss for the balanced data sets - CNN model

Figure 6.29 shows the loss values of the GRU model on the original data sets. The loss values are 0.152 on the *ant* data set, 0.146 on the *camel* data set, 0.076 on the *ivy* data set, 0.028 on the *jedit* data set, 0.048 on the *log4j* data set, and 0.090 on the *xerces* data set.

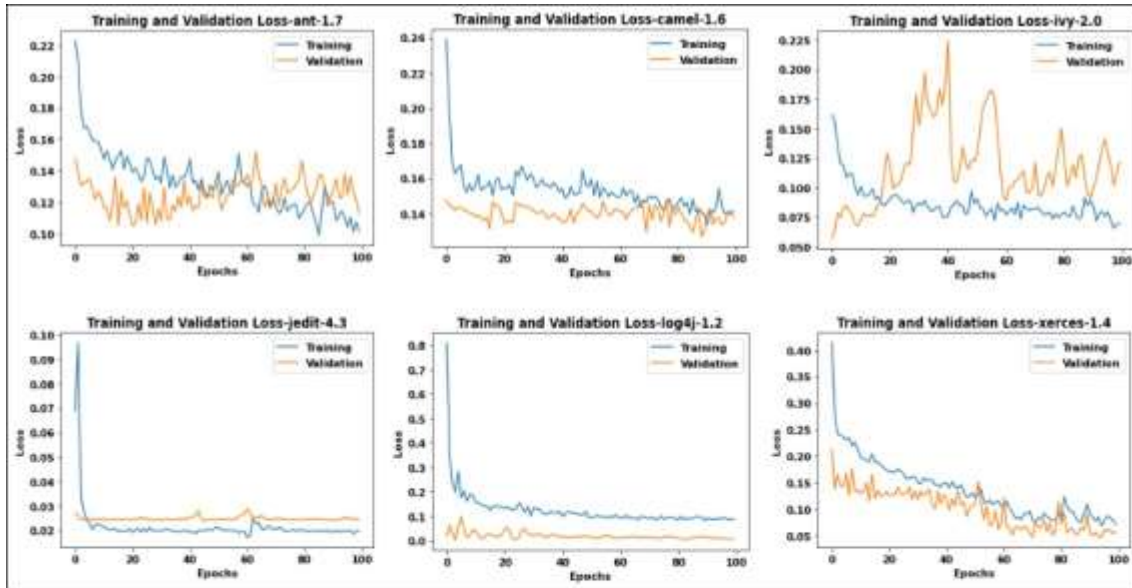


Figure 6.29 Training and Validation Loss for the original data sets - GRU model

Figure 6.30 shows the loss values of the GRU model on the balanced data sets. The loss values are 0.130 on the *ant* data set, 0.144 on the *camel* data set, 0.055 on the *ivy* data set, 0.026 on the *jedit* data set, 0.073 on the *log4j* data set, and 0.064 on the *xerces* data set. As shown in the Figures, the accuracy of training and validation increases, and the loss decreases with increasing epochs. Regarding the high accuracy and low loss obtained by the proposed models, we note that the models are well-trained and validated.

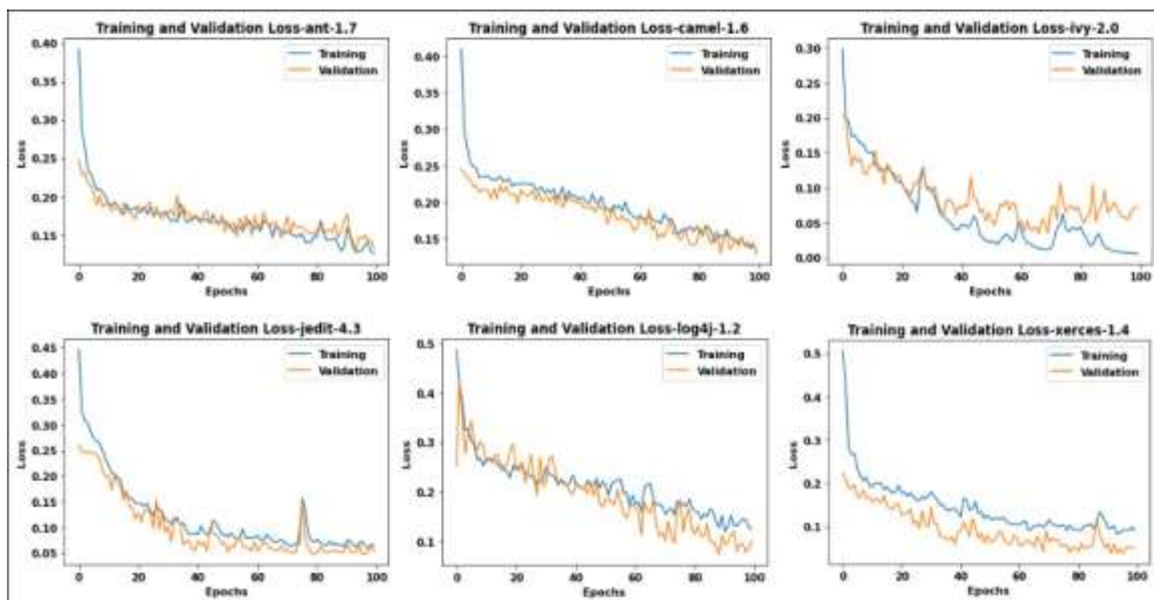


Figure 6.30 Training and Validation Loss for the balanced data sets - GRU model

Figures 6.31 to 6.34 show the ROC curves of the models on the original and balanced datasets. The vertical axis presents the actual positive rate of the model, and the horizontal axis illustrates the false positive rate. The AUC is a sign of the performance of the model. The larger the AUC is, the better the model performance will be. Based on the Figures, the values are encouraging and indicate our proposed model's efficiency in SDP.

Figure 6.31 shows the AUC values of the CNN model on the original data sets. The best AUC obtained is 95% on the *xerces* data set, while the worst AUC is 46% on the *log4j* data set.

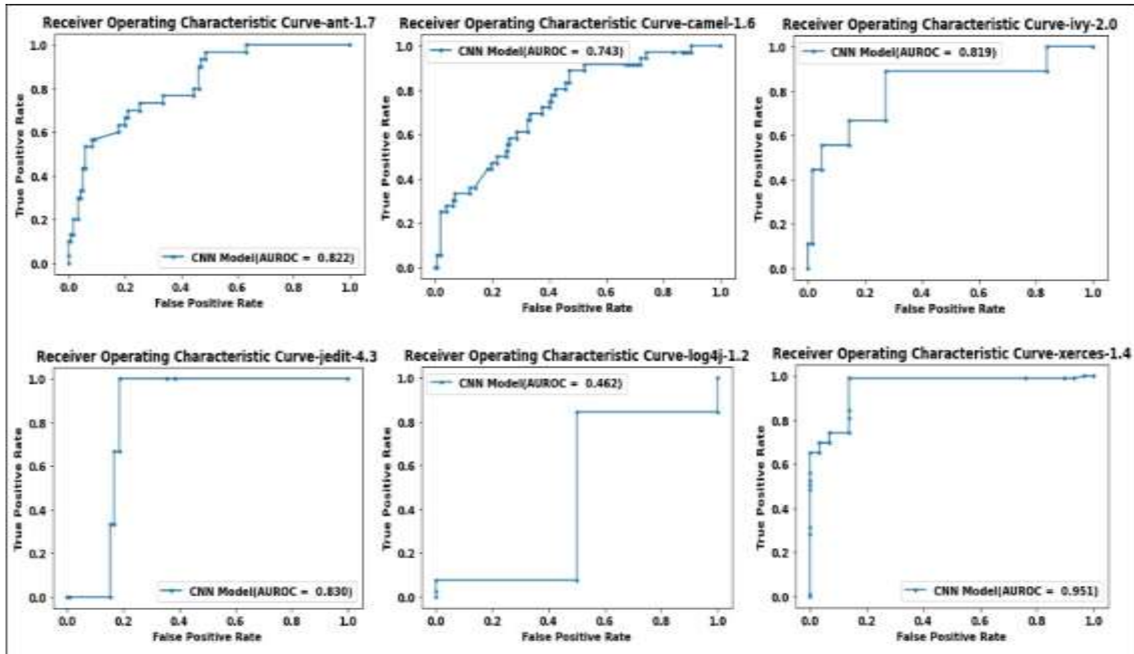


Figure 6.31 ROC curves for the original data sets - CNN model

Figure 6.32 shows the AUC values of the CNN model on the balanced data sets. The best AUC obtained is 99% on the *log4j* and *xerces* data sets, while the worst AUC is 90% on the *camel* data set.

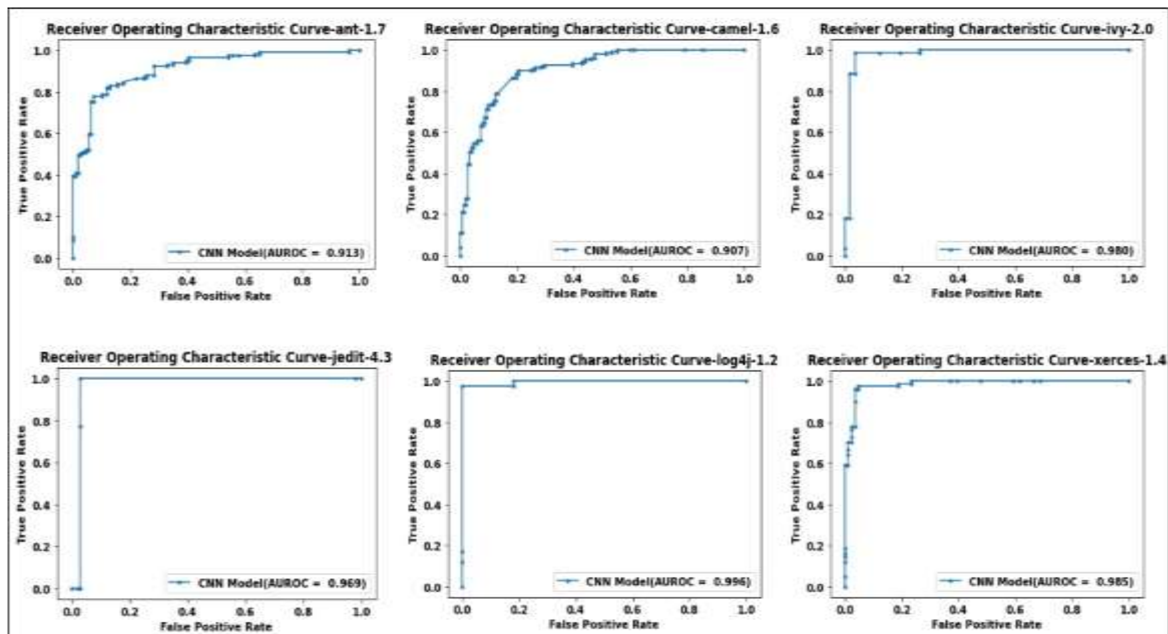


Figure 6.32 ROC curves for the balanced data sets - CNN model

Figure 6.33 shows the AUC values of the GRU model on the original data sets. The best AUC obtained is 93% on the *jedit* data set, while the worst AUC is 29% on the *log4j* data set.

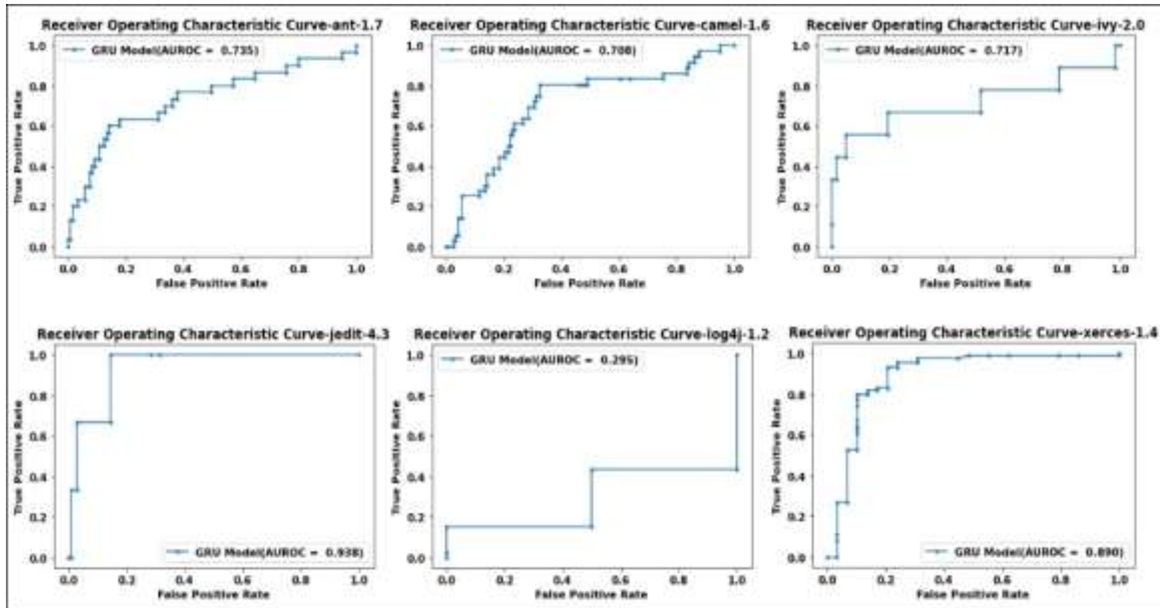


Figure 6.33 ROC curves for the original data sets - GRU model

Figure 6.34 shows the AUC values of the GRU model on the balanced data sets. The best AUC obtained is 100% on the *jedit* data set, while the worst AUC is 87% on the *camel* data set. Further, in appendix 3, Figures 1 to 4 show the AUCPR of the models on the original and balanced datasets.

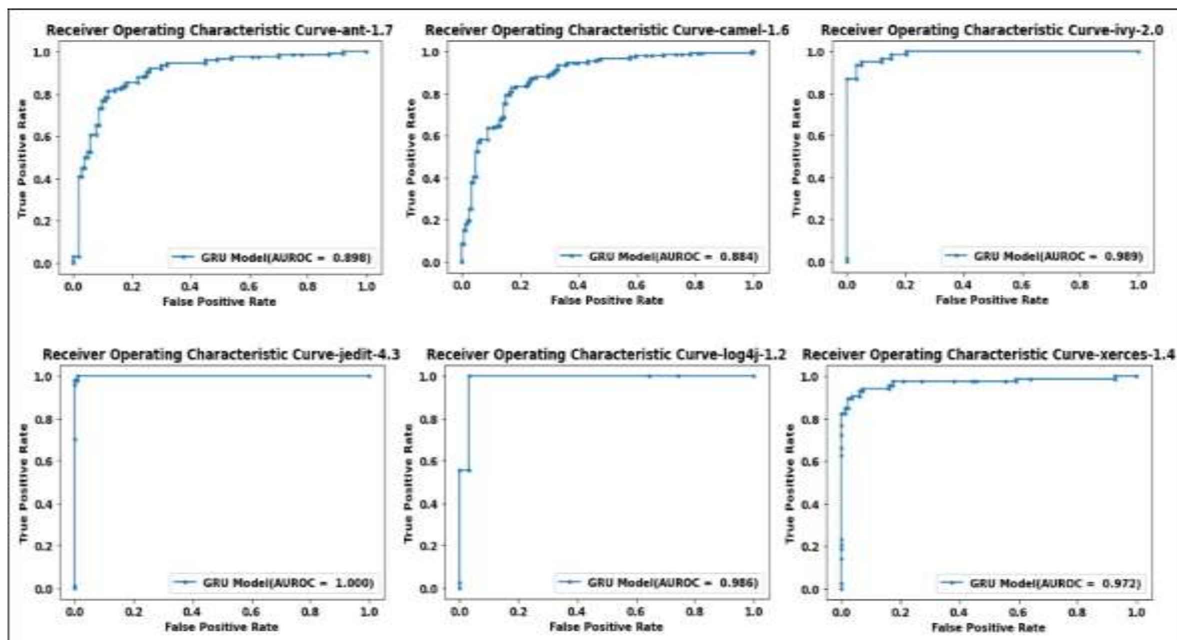


Figure 6.34 ROC curves for the balanced data sets - GRU model

Table 6.24 shows the comparison of the results produced using our models with those obtained using the baseline model (RF) based on six performance measures: accuracy precision, recall, f-Measure, MCC, and AUC. According to Table 6.24, our models outperform the baseline model in some datasets. We also compared the results produced using our models with those obtained in previous studies based on six performance measures: accuracy precision, recall, f-Measure, MCC, and AUC. Table 6.25 compares the performance measures obtained by our models and the performance values in previous studies. The best values are indicated with bold

text and "- "to indicate the approaches that did not provide results in a particular data set. According to Table 6.25, some of the results in the previous studies are better than ours. Still, in most cases, our models outperform the state-of-the-art approaches and provide better predictive performance.

Table 6.24 Performance measures of the baseline model (RF) and proposed models

Models	Datasets	Performance Measures					
		Accuracy	**Precision**	**Recall**	**F-Measure**	**MCC**	**AUC**
RF	ant	0.83	0.57	0.57	0.57	0.45	0.72
	camel	0.82	0.56	0.28	0.37	0.30	0.61
	ivy	0.90	0.67	0.44	0.53	0.49	0.70
	jedit	0.97	0.00	0.00	0.00	0.00	0.50
	log4j	0.98	0.97	1.00	0.99	0.69	0.75
	xerces	0.95	0.95	0.99	0.97	0.86	0.90
Averages		**0.90**	**0.62**	**0.54**	**0.57**	**0.46**	**0.69**
CNN with SMOTE Tomek	ant	0.85	0.87	0.82	0.85	0.69	0.91
	camel	0.84	0.81	0.90	0.85	0.69	0.90
	ivy	0.95	0.92	0.98	0.95	0.90	0.98
	jedit	0.97	0.94	1.00	0.97	0.93	0.96
	log4j	0.97	0.98	0.98	0.98	0.94	0.99
	xerces	0.95	0.93	0.98	0.95	0.90	0.98
Averages		**0.92**	**0.90**	**0.94**	**0.92**	**0.84**	**0.95**
GRU with SMOTE Tomek	ant	0.83	0.88	0.81	0.85	0.67	0.89
	camel	0.82	0.82	0.82	0.82	0.63	0.87
	ivy	0.95	0.95	0.95	0.95	0.90	0.98
	jedit	0.99	0.98	1.00	0.99	0.97	1.00
	log4j	0.96	0.98	0.95	0.96	0.91	0.98
	xerces	0.93	0.92	0.94	0.93	0.85	0.97
Averages		**0.91**	**0.92**	**0.91**	**0.91**	**0.82**	**0.94**

Table 6.25 Comparison of the proposed models with other existing approaches

Approaches	Datasets	Performance Measures					
		Accuracy	**Precision**	**Recall**	**F-Measure**	**MCC**	**AUC**
Hybrid Neural Network model[46]	JEdit, IVY, Ant, Camel	0.97, 0.88, 0.81, 0.81	**1.00**, 0.99, 0.93, **1.00**	**1.00**, 0.88, 0.84, 0.81	0.98, 0.93, 0.88, 0.89	-	-
LSTM[48]	Camel	-	0.51	0.41	0.46	-	-
CNN[80]	ant, camel, ivy, jedit, log4j	-	-	-	0.39, 0.52, 0.31, 0.00, 0.97	0.30, 0.42, 0.25, 0.00, 0.00	-
BPDET[82]	CM1, JM1, KC1, MC1, PC1, MW1	-	-	-	0.84, 0.76, 0.83, 0.96, 0.92, 0.90	0.42, 0.23, 0.33, 0.14, 0.38, 0.33	0.75, 0.75, 0.81, 0.85, 0.88, 0.77
DP-ARNN[84]	Camel, Xerces, JEdit	-	-	-	0.51, 0.27, 0.56	-	0.79, 0.76, 0.82
RF[87]	ant, camel, ivy, jedit	-	-	-	-	0.42, 0.20, 0.24, 0.26	-
DT[87]	ant, camel, ivy, jedit	-	-	-	-	0.29, 0.18, 0.20, 0.12	-
LR[96]	Ant, Camel, IVY	-	-	-	0.52, 0.34, 0.30	-	-
K-NN[96]	Ant, Camel, IVY	-	-	-	0.53, 0.37, 0.30	-	-
MLP[96]	Ant, Camel, IVY	-	-	-	0.50, 0.38, 0.25	-	-

SVM[96]	Ant, Camel, IVY	-	-	-	0.50, 0.084, 0.28	-	-
CBIL[103]	Camel, JEdit, Xerces	-	-	-	0.93, 0.85, 0.95	-	0.96, 0.91, 0.98
DP-LSTM[104]	Camel, Jedit, Log4j, Xerces	-	-	-	0.37, 0.44, 0.52, 0.26	-	-
HyGRAR[106]	JEdit, Ant	0.98, 0.96	0.70, 0.98	0.63, 0.85	-	-	0.81, 0.92
SPFCNN[107]	CM1, JM1, KC1, PC1, MW1	-	-	-	-	0.85, 0.74, 0.78, 0.87, 0.80	0.92, 0.87, 0.88, 0.93, 0.90
CNN with SMOTE Tomek	ant, camel, ivy, jedit, log4j, xerces	0.85, 0.84, 0.95, 0.97, 0.97, 0.95	0.87, 0.81, 0.92, 0.94, 0.98, 0.93	0.82, 0.90, 0.98, **1.00**, 0.98, 0.98	0.85, 0.85, 0.95, 0.97, 0.98, 0.95	0.69, 0.69, 0.90, 0.93, 0.94, 0.90	0.91, 0.90, 0.98, 0.96, 0.99, 0.98
GRU with SMOTE Tomek	ant, camel, ivy, jedit, log4j, xerces	0.83, 0.82, 0.95, **0.99**, 0.96, 0.93	0.88, 0.82, 0.95, 0.98, 0.98, 0.92	0.81, 0.82, 0.95, **1.00**, 0.95, 0.94	0.85, 0.82, 0.95, **0.99**, 0.96, 0.93	0.67, 0.63, 0.90, **0.97**, 0.91, 0.85	0.89, 0.87, 0.98, **1.00**, 0.98, 0.97

In summary, this study aimed to propose a novel SDP approach based on CNN and GRU combined with a hybrid sampling method (SMOTE Tomek) for SDP. We compared the results obtained by the proposed approach based on the original and balanced datasets to investigate the impact of hybrid sampling methods on improving the accuracy of ML techniques. Additionally, the proposed approach's results were compared with those presented in previous studies. After comparing the results obtained by the proposed models on the original datasets with results obtained by the proposed models on the balanced datasets, as shown in the Tables and Figures, we note that the models got good scores on the balanced datasets and the results improved further due to balancing, which indicated that the combination of CNN and GRU with hybrid sampling method (SMOTE Tomek) has a positive effect on the performance of SDP regarding datasets with imbalanced class distributions. Furthermore, data sampling methods play an essential role in improving the accuracy of the ML models in predicting software defects. Regarding the evaluation of the results obtained from our proposed approach and their comparison with some results of other studies, we conclude that our models are promising in predicting software defects and outperform other models in the previous studies.

6.5 Summary

In this chapter, we presented the experimental results and discussion of software bugs prediction. The experimental results have been compared and evaluated based on several standard performance measures. We compared experimental results based on the original and balanced datasets and compared our results with current state-of-the-art results for the prediction of software bugs. The results showed that our proposed methods significantly outperform current state-of-the-art methods for predicting software bugs. We concluded that the combined data-balancing methods with ML techniques significantly enhance the accuracy of predicting software bugs. We observe that the incorporation of appropriate data-balancing methods and ML techniques not only enhances the model's ability to predict software bugs accurately but also mitigates the bias towards the majority class, resulting in a more balanced performance across different classes of software bugs. This research has practical implications for software developers and researchers. It highlights the significance of considering data-balancing methods when applying ML models for predicting software bugs. By employing these methods, developers can enhance their ability to identify and address code quality issues, thereby improving software maintainability.

Chapter 7 Experimental Results and Discussion of Code Smells Detection

This subsection presents the results obtained from the experiments explained in the previous section (proposed methodology and implementation) which includes the results of code smells detection.

7.1 ML techniques with Oversampling Methods in Code Smells Detection

In this sub-section, we discuss the findings of the fifth study. The aim was to present a method based on five ML models, namely DT, K-NN, SVM, XGB, and MLP combined with Oversampling method (Random Oversampling) to detect four code smells (God class, data class, long method, and feature envy). The experiments have been conducted based on benchmark datasets obtained from the Qualitas Corpus Systems. The experimental results were evaluated and compared based on various performance measures (accuracy, precision, recall, f-measure, MCC, and AUC).

The performance of the prediction models is reported in Tables 7.1 to 7.6, and Figures 7.1 to 7.4.

Tables 7.1 to 7.4 present model results based on the original and balanced datasets. Based on the DT model, we observed that accuracy values varied from 0.92 to 0.99 on the original datasets and from 0.98 to 1.00 on the balanced datasets. In terms of precision, the values ranged from 0.86 to 1.00 on the original datasets and from 0.97 to 1.00 on the balanced datasets. The recall values ranged from 0.89 to 0.96 on the original datasets and were 1.00 on the balanced datasets. In the context of f-measure, the values varied from 0.87 to 0.98 on the original datasets and from 0.98 to 1.00 on the balanced datasets. Moreover, MCC values ranged from 0.81 to 0.97 on the original datasets and from 0.96 to 1.00 on the balanced datasets, whereas AUC values ranged from 0.90 to 0.98 on the original datasets and from 0.98 to 1.00 on the balanced datasets.

The K-NN model demonstrates that the accuracy values vary between 0.86 to 0.92 on the original datasets and from 0.91 to 0.97 on the balanced datasets. Additionally, the precision values on the original datasets vary from 0.75 to 0.97 and from 0.88 to 0.97 on the balanced datasets. The recall values vary from 0.70 to 0.91 on the original datasets and from 0.97 to 0.98 on the balanced datasets. In the context of f-measure, the values range from 0.76 to 0.88 on the original datasets and from 0.92 to 0.98 on the balanced datasets. Furthermore, the MCC values range from 0.66 to 0.81 on the original datasets and from 0.82 to 0.94 on the balanced datasets. Finally, the AUC values range from 0.85 to 0.97 on the original datasets and from 0.93 to 0.98 on the balanced datasets.

Following the SVM model, it can be observed that the accuracy values vary between 0.90 and 0.98 on the original datasets, and from 0.96 to 1.00 on the balanced datasets. On the original datasets, the precision values vary from 0.85 to 0.96, while on the balanced datasets, the precision values vary from 0.94 to 1.00. In the context of recall, the values range from 0.85 to 0.96 on the original datasets, and from 0.98 to 1.00 on the balanced datasets. In the context of f-measure, the values range from 0.85 to 0.96 on the original datasets and from 0.97 to 1.00 on the balanced datasets. The MCC values range from 0.78 to 0.94 on the original datasets and from 0.92 to 1.00 on the balanced datasets. The AUC values range from 0.96 to 0.99 on the original datasets and from 0.97 to 1.00 on the balanced datasets.

Based on the XGB model, it can be observed that the accuracy values range between 0.95 to 1.00 for the original datasets and between 0.96 to 1.00 for the balanced datasets. In the context of precision, the values range between 0.87 to 1.00 for the original datasets and between 0.95

to 1.00 for the balanced datasets. In the context of recall, the values range between 0.97 to 1.00 for the original datasets and between 0.97 to 1.00 for the balanced datasets. In the context of f-measure, the values range between 0.93 to 1.00 for the original datasets and between 0.96 to 1.00 for the balanced datasets. Additionally, the MCC values range between 0.89 to 1.00 for the original datasets and between 0.90 to 1.00 for the balanced datasets, whereas the AUC values range between 0.99 to 1.00 for the original datasets and between 0.98 to 1.00 for the balanced datasets.

Based on the MLP model, it was observed that the accuracy values ranged from 0.88 to 0.98 on the original datasets and from 0.96 to 0.98 on the balanced datasets. Furthermore, the precision values ranged from 0.87 to 0.97 on the original datasets and from 0.96 to 0.97 on the balanced datasets, while the recall values ranged from 0.74 to 1.00 on the original datasets and from 0.97 to 1.00 on the balanced datasets. In the context of f-measure, the values ranged from 0.80 to 0.96 on the original datasets and from 0.97 to 0.98 on the balanced datasets. Furthermore, the MCC values range from 0.72 to 0.94 on the original datasets and from 0.92 to 0.96 on the balanced datasets. Finally, the AUC values range from 0.90 to 0.99 on the original datasets and from 0.98 to 1.00 on the balanced datasets.

Concerning each type of code smell, the top-performing models attain the subsequent results: DT model scores 100% accuracy on data class and long method (balanced datasets). K-NN model achieves 97% accuracy on God class (balanced datasets). The SVM model scores 100% accuracy on the long method (balanced datasets). XGB model achieves 100% accuracy on data class and long method (original and balanced datasets). MLP model scores 98% accuracy on data class (original and balanced datasets) and 98% on the long method (balanced datasets).

Table 7.1 Evaluation Results for the Class-Level Dataset: God class_ original and balanced datasets

Original datasets						
ML Models	**Performance measurement**					
	Accuracy	**Precision**	**Recall**	**F- measure**	**MCC**	**AUC**
DT	0.95	0.97	0.92	0.94	0.90	0.94
K-NN	0.90	0.97	0.81	0.88	0.81	0.94
SVM	0.92	0.94	0.86	0.90	0.83	0.97
XGB	0.98	0.97	0.97	0.97	0.95	0.99
MLP	0.93	0.97	0.86	0.91	0.85	0.99
Averages	**0.93**	**0.96**	**0.88**	**0.92**	**0.86**	**0.96**
Balanced datasets						
ML Models	**Performance measurement**					
	Accuracy	**Precision**	**Recall**	**F- measure**	**MCC**	**AUC**
DT	0.98	0.97	1.00	0.98	0.96	0.98
K-NN	0.97	0.97	0.98	0.98	0.94	0.97
SVM	0.96	0.95	0.98	0.97	0.92	0.99
XGB	0.96	0.95	0.97	0.96	0.90	0.98
MLP	0.97	0.97	0.98	0.98	0.94	0.98
Averages	**0.96**	**0.96**	**0.98**	**0.97**	**0.93**	**0.98**

Table 7.2 Evaluation Results for the Class-Level Dataset: Data class_ original and balanced datasets

Original datasets						
ML Models	**Performance measurement**					
	Accuracy	**Precision**	**Recall**	**F- measure**	**MCC**	**AUC**
DT	0.98	1.00	0.91	0.95	0.94	0.95
K-NN	0.89	0.75	0.91	0.82	0.75	0.97
SVM	0.96	0.92	0.96	0.94	0.91	0.99
XGB	1.00	1.00	1.00	1.00	1.00	1.00
MLP	0.98	0.92	1.00	0.96	0.94	0.99

Averages	0.96	0.91	0.95	0.93	0.90	0.98
Balanced datasets						
ML Models	Performance measurement					
	Accuracy	Precision	Recall	F- measure	MCC	AUC
DT	1.00	1.00	1.00	1.00	1.00	1.00
K-NN	0.96	0.93	0.98	0.96	0.91	0.98
SVM	0.97	0.95	1.00	0.97	0.94	0.99
XGB	1.00	1.00	1.00	1.00	1.00	1.00
MLP	0.98	0.97	1.00	0.98	0.96	0.99
Averages	**0.98**	**0.97**	**0.99**	**0.98**	**0.96**	**0.99**

Table 7.3 Evaluation Results for the Method-Level Dataset: Long method_ original and balanced datasets

Original datasets						
ML Models	Performance measurement					
	Accuracy	Precision	Recall	F- measure	MCC	AUC
DT	0.99	1.00	0.96	0.98	0.97	0.98
K-NN	0.92	0.92	0.81	0.86	0.80	0.94
SVM	0.98	0.96	0.96	0.96	0.94	0.99
XGB	1.00	1.00	1.00	1.00	1.00	1.00
MLP	0.94	0.87	0.96	0.91	0.87	0.98
Averages	**0.96**	**0.95**	**0.93**	**0.94**	**0.91**	**0.97**
Balanced datasets						
ML Models	Performance measurement					
	Accuracy	Precision	Recall	F- measure	MCC	AUC
DT	1.00	1.00	1.00	1.00	1.00	1.00
K-NN	0.96	0.93	0.98	0.95	0.91	0.97
SVM	1.00	1.00	1.00	1.00	1.00	1.00
XGB	1.00	1.00	1.00	1.00	1.00	1.00
MLP	0.98	0.96	1.00	0.98	0.96	1.00
Averages	**0.98**	**0.97**	**0.99**	**0.98**	**0.97**	**0.99**

Table 7.4 Evaluation Results for the Method-Level Dataset: Feature envy_ original and balanced datasets

Original datasets						
ML Models	Performance measurement					
	Accuracy	Precision	Recall	F- measure	MCC	AUC
DT	0.92	0.86	0.89	0.87	0.81	0.90
K-NN	0.86	0.83	0.70	0.76	0.66	0.85
SVM	0.90	0.85	0.85	0.85	0.78	0.96
XGB	0.95	0.87	1.00	0.93	0.89	0.99
MLP	0.88	0.87	0.74	0.80	0.72	0.90
Averages	**0.90**	**0.85**	**0.83**	**0.84**	**0.77**	**0.92**
Balanced datasets						
ML Models	Performance measurement					
	Accuracy	Precision	Recall	F- measure	MCC	AUC
DT	0.98	0.97	1.00	0.98	0.96	0.98
K-NN	0.91	0.88	0.97	0.92	0.82	0.93
SVM	0.96	0.94	1.00	0.97	0.92	0.97
XGB	0.98	0.97	1.00	0.98	0.96	0.98
MLP	0.96	0.97	0.97	0.97	0.92	0.98
Averages	**0.95**	**0.94**	**0.98**	**0.96**	**0.91**	**0.96**

We used Boxplots to aggregate the achieved results to get a more accurate overview of the quality of the results. Figure 7.1 exhibits box plots that display the averages of several performance measures, including accuracy, precision, recall, f-measure, MCC, and AUC based on the original datasets. The overall average performance of all models is 0.93, 0.96, 0.88, 0.92, 0.86, and 0.96, respectively, for the god class. Similarly, for the data class, the overall average

performance of all models is 0.96, 0.91, 0.95, 0.93, 0.90, and 0.98, respectively. In the context of the long method, the overall average of all models is 0.96, 0.95, 0.93, 0.94, 0.91, and 0.97, respectively. Lastly, for feature envy, the overall average performance of all models is 0.90, 0.85, 0.83, 0.84, 0.77, and 0.92, respectively.

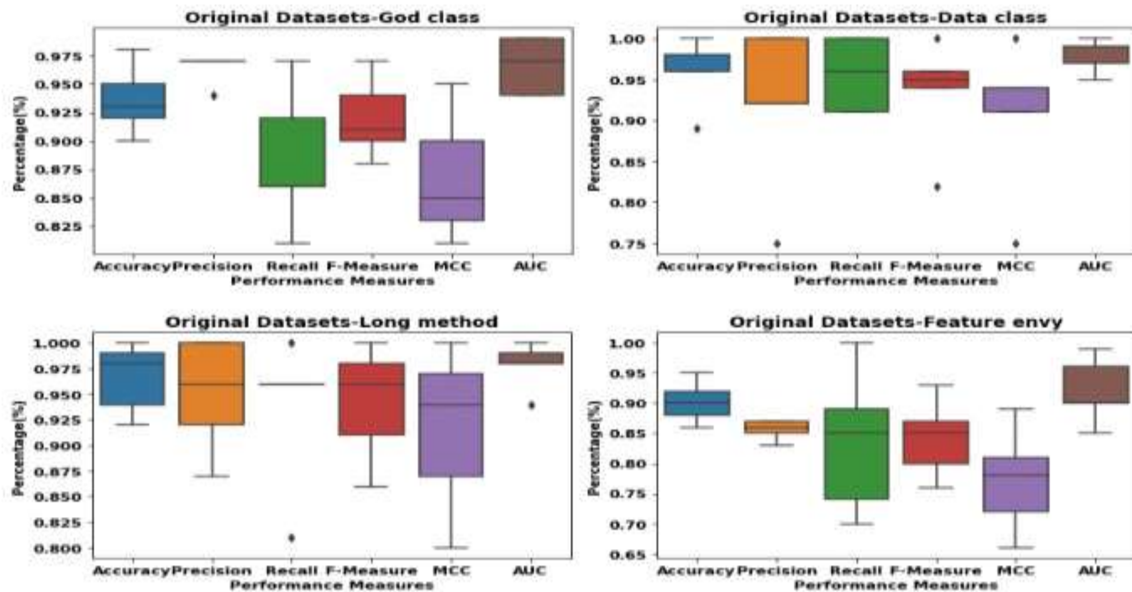


Figure 7.1 Box Plots represent the models' performance measures on all considered code smells_ original datasets

Figure 7.2 exhibits box plots that display the averages of several performance measures, including accuracy, precision, recall, f-measure, MCC, and AUC based on the balanced datasets. The overall average performance of all models is 0.96, 0.96, 0.98, 0.97, 0.93, and 0.98, respectively, for the god class. Similarly, for the data class, the overall average performance of all models is 0.98, 0.97, 0.99, 0.98, 0.96, and 0.99, respectively. In the context of the long method, the overall average of all models is 0.98, 0.97, 0.99, 0.98, 0.97, and 0.99, respectively. Lastly, for feature envy, the overall average performance of all models is 0.95, 0.94, 0.98, 0.96, 0.91, and 0.96, respectively.

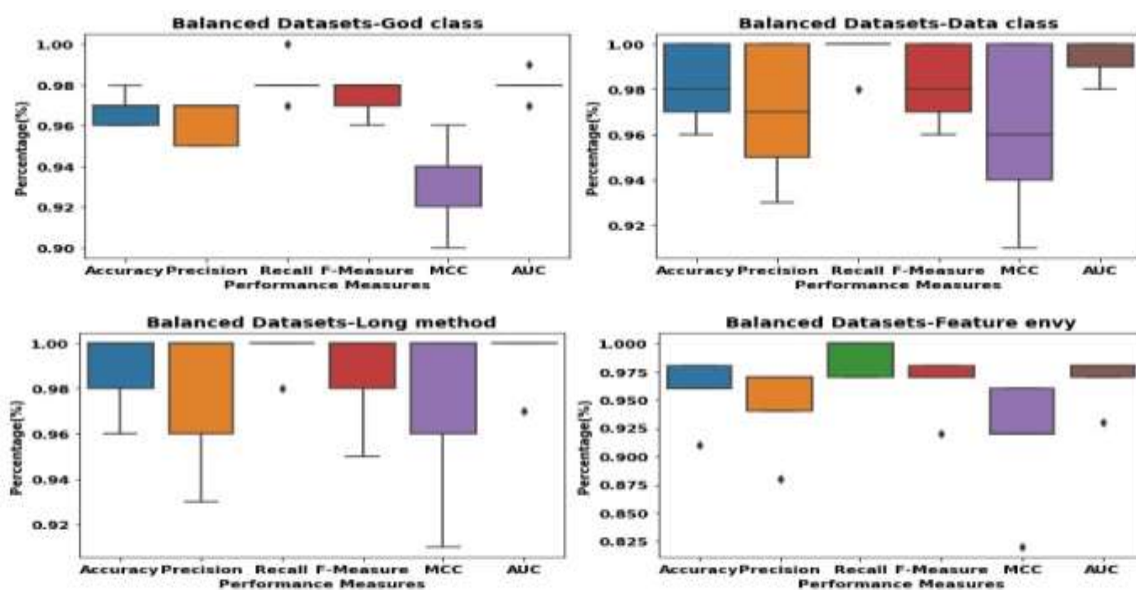


Figure 7.2 Box Plots represent the models' performance measures on all considered code smells_ balanced datasets

Figures 7.3 and 7.4 show the ROC curves of the models on the original and balanced datasets. The vertical axis presents the actual positive rate of the model, and the horizontal axis illustrates the false positive rate. The AUC is a sign of the performance of the model. The larger AUC is, the better the model performance will be. Based on the Figures, the values are encouraging and indicate our proposed model's efficiency in code smell detection.
Figure 7.3 shows the AUC of the models for all considered code smells on the original datasets; the highest AUC on the original datasets (God class) is 99%, obtained by XGB and MLP models. The the lowest AUC is 94%, obtained by DT and K-NN models. The highest AUC on the original datasets (data class) is 100% obtained by the XGB model, while the lowest AUC is 95% obtained by the DT model. The highest AUC on the original datasets (long method) is 100% obtained by the XGB model, while the lowest AUC is 94% obtained by the K-NN model. The highest AUC on the original datasets (feature envy) is 99%, obtained by the XGB model, while the lowest AUC is 85%, obtained by the K-NN model.

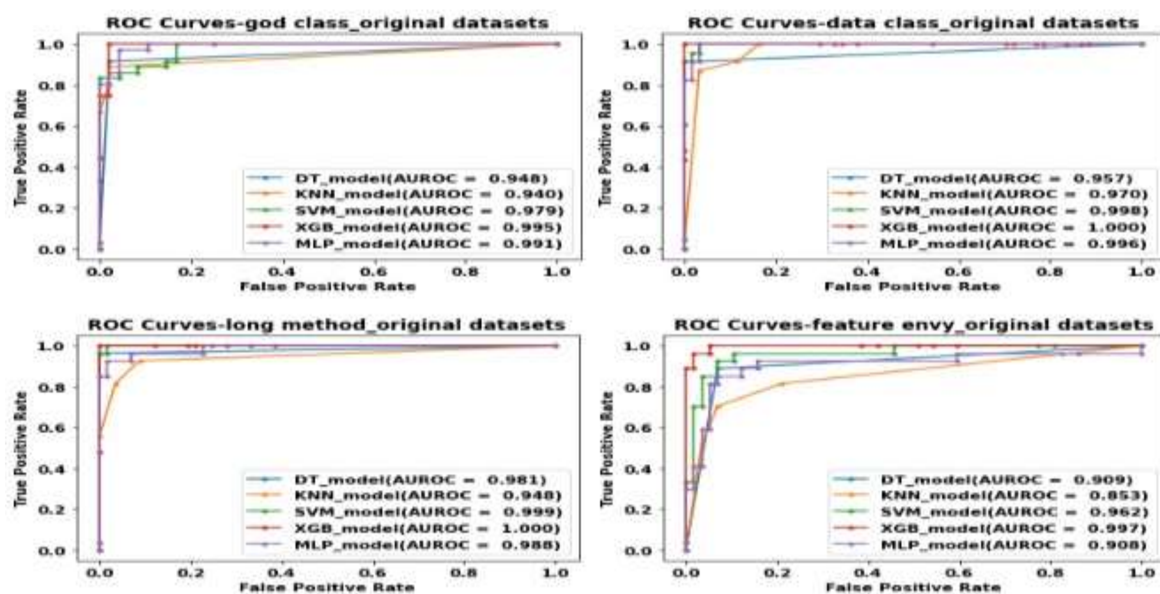


Figure 7.3 The ROC curves obtained by the models on all considered code smells_ original datasets

Figure 7.4 shows the AUC of the models for all considered code smells on the balanced datasets, the highest AUC on the balanced datasets (God class) is 99%, obtained by the SVM model, while the lowest AUC is 97%, and the K-NN model gets. The highest AUC on the balanced datasets (data class) is 100% obtained by DT and XGB models, while the lowest AUC is 98% obtained by the K-NN model. The highest AUC on the balanced datasets (long method) is 100% acquired by DT, SVM, XGB, and MLP models, while the lowest AUC is 97%, which the K-NN model obtains. The highest AUC on the balanced datasets (feature envy) is 99%, obtained by DT, XGB, and MLP models, while the lowest AUC is 93% which the K-NN model gets.

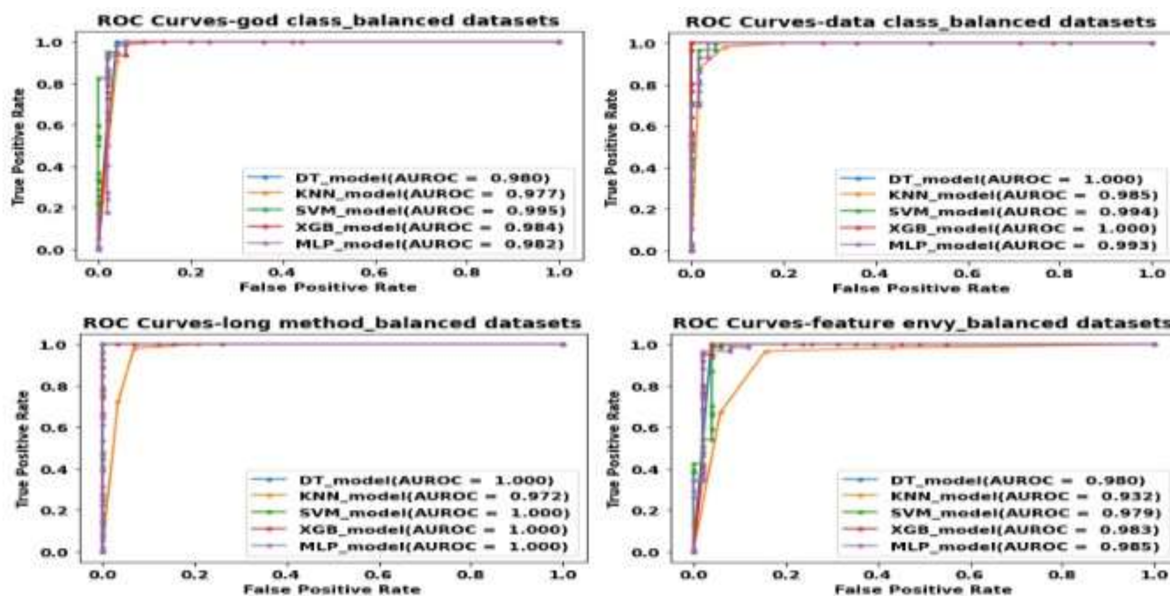


Figure 7.4 The ROC curves obtained by the models on all considered code smells_ balanced datasets

Tables 7.5 and 7.6 show the comparison results of our method with some previous studies based on some performance measures, namely accuracy and AUC. The best values are indicated in bold, and " - " denotes the missing performance measures for specific methods in certain datasets. Overall, our method outperforms the other state-of-the-art methods in most cases.

Table 7.5 Comparison of the proposed method with other existing methods based on the accuracy

Methods	**Datasets**			
	God class	**Data class**	**Long method**	**Feature envy**
RF[4]	0.96	0.98	0.99	0.96
NB[4]	0.97	0.97	0.97	0.91
DT[27]	-	-	-	0.97
RF[27]	-	0.99	0.95	-
K-NN[108]	0.97	0.97	0.97	0.91
NB[108]	0.96	0.84	0.95	0.92
MLP[108]	0.97	0.97	0.96	0.95
DT[108]	0.97	0.98	0.98	0.98
RF[108]	0.97	0.98	0.99	0.97
LR[108]	0.97	0.97	0.99	0.97
NB[118]	0.96	-	0.97	0.91
MLP[118]	0.97	-	0.99	0.92
DT[118]	0.98	-	0.97	0.95
RF[119]	0.76	0.81	0.60	0.66
NB[119]	0.74	0.66	0.74	0.76
SVM[119]	0.66	0.66	0.66	0.60
RF[120]	0.69	0.70	0.68	0.71
NB[120]	0.82	0.75	0.81	0.83
SVM[120]	0.74	0.83	0.81	0.83
K-NN[120]	0.80	0.82	0.81	0.82
Our models (DT, K-NN, SVM, XGB, MLP) - Original Datasets	0.95, 0.90, 0.92, **0.98**, 0.93	0.98, 0.89, 0.96, **1.00**, 0.98	0.99, 0.92, 0.98, **1.00**, 0.94	0.92, 0.86, 0.90, 0.95, 0.88
Our models (DT, K-NN, SVM, XGB, MLP) - Balanced Datasets	**0.98**, 0.97, 0.96, 0.96, 0.97	**1.00**, 0.96, 0.97, **1.00**, 0.98	**1.00**, 0.96, **1.00**, **1.00**, 0.98	**0.98**, 0.91, 0.96, **0.98**, 0.96

Table 7.6 Comparison of the proposed method with other existing methods based on AUC

Methods	Datasets			
	God class	**Data class**	**Long method**	**Feature envy**
RF[120]	0.59	0.65	0.52	0.59
NB[120]	0.88	0.85	0.86	0.86
SVM[120]	0.65	0.88	0.66	0.82
K-NN[120]	0.83	0.86	0.86	0.83
Our models (DT, K-NN, SVM, XGB, MLP) - Original Datasets	0.94, 0.94, 0.97, **0.99**, **0.99**	0.95, 0.97, 0.99, **1.00**, 0.99	0.98, 0.94, 0.99, **1.00**, 0.98	0.90, 0.85, 0.96, **0.99**, 0.90
Our models (DT, K-NN, SVM, XGB, MLP) - Balanced Datasets	0.98, 0.97, **0.99**, 0.98, 0.98	**1.00**, 0.98, 0.99, **1.00**, 0.99	**1.00**, 0.97, **1.00**, **1.00**, **1.00**	0.98, 0.93, 0.97, 0.98, 0.98

In summary, this study aimed to present a method based on five ML models, namely DT, K-NN, SVM, XGB, and MLP combined with Oversampling method (Random Oversampling) to detect code smells. We compared the results obtained by the proposed method based on the original and balanced datasets to investigate the impact of Oversampling methods on improving the accuracy of ML techniques. Additionally, the proposed method's results were compared with those presented in previous studies. After comparing the results obtained by the proposed models on the original datasets with results obtained by the proposed models on the balanced datasets, as shown in the Tables and Figures, we note that the models got good scores on the balanced datasets and the results improved further due to balancing, which indicated that the combination of DT, K-NN, SVM, XGB, and MLP with Oversampling method (Random Oversampling) has positive effect on the performance of code smells detection regarding datasets with imbalanced class distributions. Furthermore, data sampling methods play an essential role in improving the accuracy of the ML models in code smell detection. Regarding the evaluation of the results obtained from our proposed method and their comparison with some results of other studies, we conclude that our models are promising in code smell detection and outperform other models in the previous studies.

7.2 A Convolutional Neural Network (CNN) with Oversampling Methods

In this sub-section, we discuss the findings of the sixth study. The objective was to present a method based on a CNN with the Oversampling method (SMOTE) to detect four code smells (God class, data class, feature envy, and long method). The experiments have been conducted based on benchmark datasets obtained from the Qualitas Corpus Systems. The experimental results were evaluated and compared based on various performance measures (accuracy, precision, recall, and f-measure).

The performance of the prediction models is reported in Tables 7.7, 7.8 and 7.9, and Figures 7.5 to 7.9.

Tables 7.7 and 7.8 show the performance of the proposed model in the four code smells based on the original and balanced data sets.

- Accuracy for the four code smell datasets: The proposed model using the balanced datasets achieves greater accuracy than the proposed model using the original datasets on the Feature Envy and Long Method datasets, which are 98 % and 100%. The lowest accuracy was achieved by the proposed model using the original datasets on the Feature Envy dataset by up to 95%.

- Precision for the four code smell datasets: The proposed model using the balanced datasets achieves greater precision than the proposed model using the original datasets on the Feature Envy and Long Method datasets, which are 98 % and 100%. The proposed model achieved the

lowest precision using the original datasets on the Feature Envy and Long Method datasets by up to 93%.

- Recall for the four code smell datasets: The proposed model using the balanced datasets achieves more excellent recall than the proposed model using the original datasets on the God Class, Data Class, and Feature Envy datasets, which are 97%, 100 %, and 98%. The lowest recall was achieved by the proposed model using the original datasets on the Feature Envy dataset by up to 93%.

- F-Measure for the four code smell datasets: The proposed model using the balanced datasets achieves greater F-Measure than the proposed model using the original datasets on the God Class, Feature Envy, and Long Method datasets, which are 97%, 98%, and 100%. The proposed model achieved the lowest F-Measure using the original datasets on the Feature Envy dataset by up to 93%.

Table 7.7 Performance analysis for proposed CNN Model - Original Datasets

Original Datasets	**Performance Measures**			
	Accuracy	**Precision**	**Recall**	**F-Measure**
God Class	0.96	0.97	0.94	0.96
Data Class	0.99	1.00	0.96	0.98
Feature Envy	0.95	0.93	0.93	0.93
Long Method	0.98	0.93	1.00	0.96
Averages	**0.97**	**0.95**	**0.95**	**0.95**

Table 7.8 Performance analysis for proposed CNN Model - Balanced Datasets

Balanced Datasets using SMOTE method	**Performance Measures**			
	Accuracy	**Precision**	**Recall**	**F-Measure**
God Class	0.96	0.97	0.97	0.97
Data Class	0.98	0.97	1.00	0.98
Feature Envy	0.98	0.98	0.98	0.98
Long Method	1.00	1.00	1.00	1.00
Averages	**0.98**	**0.98**	**0.98**	**0.98**

We used Boxplots to aggregate the achieved results to get a more accurate overview of the quality of the results. Figure 7.5 shows the Box plots for the performance measures (Accuracy, Precision, Recall, and F-measure) on the original and balanced datasets.

Concerning the original datasets, the highest accuracy is 99% on the Data Class dataset and the lowest accuracy is 95% on the Feature Envy dataset, the highest precision is 100% on the Data Class dataset and the lowest precision is 93% on the Feature envy and Long Method datasets, the highest recall is 100% on the Long method dataset and the lowest recall is 93% on the Feature Envy dataset, the highest f-measure is 98% on the Data Class dataset and the lowest f-measure is 93% on the Feature envy dataset.

Concerning the balanced datasets, the highest accuracy is 100% on the Long Method dataset and the lowest accuracy is 96% on the God Class dataset, the highest precision is 100% on the Long Method dataset and the lowest precision is 97% on the God Class and Data Class datasets, the highest recall is 100% on the Data Class and Long method datasets and the lowest recall is 97% on the God Class dataset, the highest f-measure is 100% on the Long Method dataset and the lowest f-measure is 97% on the God Class dataset.

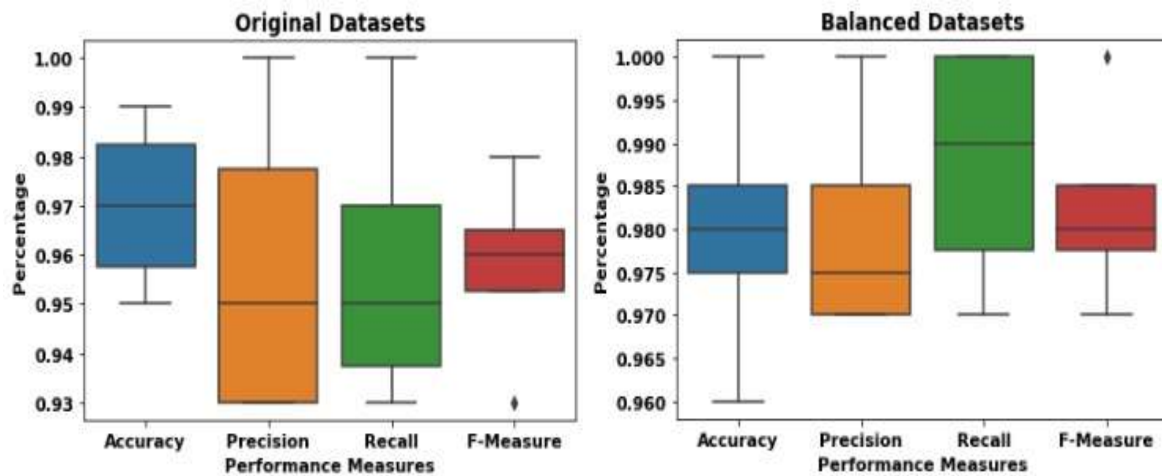


Figure 7.5 Boxplots represent performance measures obtained by CNN Model

Figures 7.6 to 7.9 show the training and validation accuracy and training and validation loss of the model on the original and balanced datasets. Figures 7.6 and 7.7 show the training and validation accuracy of the model on the original and balanced datasets. The vertical axis presents the model's accuracy, and the horizontal axis illustrates the number of epochs. Accuracy is the fraction of predictions that our model predicted right.

Figure 7.6 shows the accuracy values of the model on the original datasets. From the Figure, the model learned 96% accuracy for the God Class dataset, 99% accuracy for the Data Class dataset, 95% accuracy for the Feature Envy dataset, and 98% accuracy for the Long Method dataset at the 100th epoch.

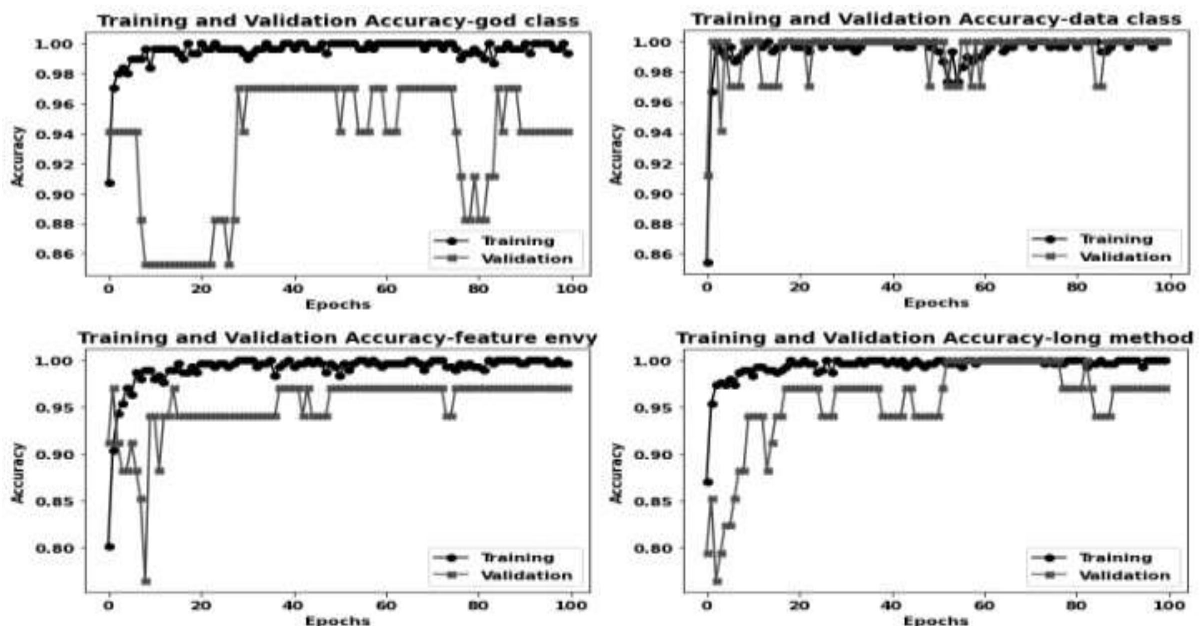


Figure 7.6 Training and Validation Accuracy over original datasets

Figure 7.7 shows the accuracy values of the model on the balanced datasets. From the Figure, the model learned 96% accuracy for the God Class dataset, 98% accuracy for the Data Class dataset, 98% accuracy for the Feature Envy dataset, and 100% accuracy for the Long Method dataset at the 100th epoch.

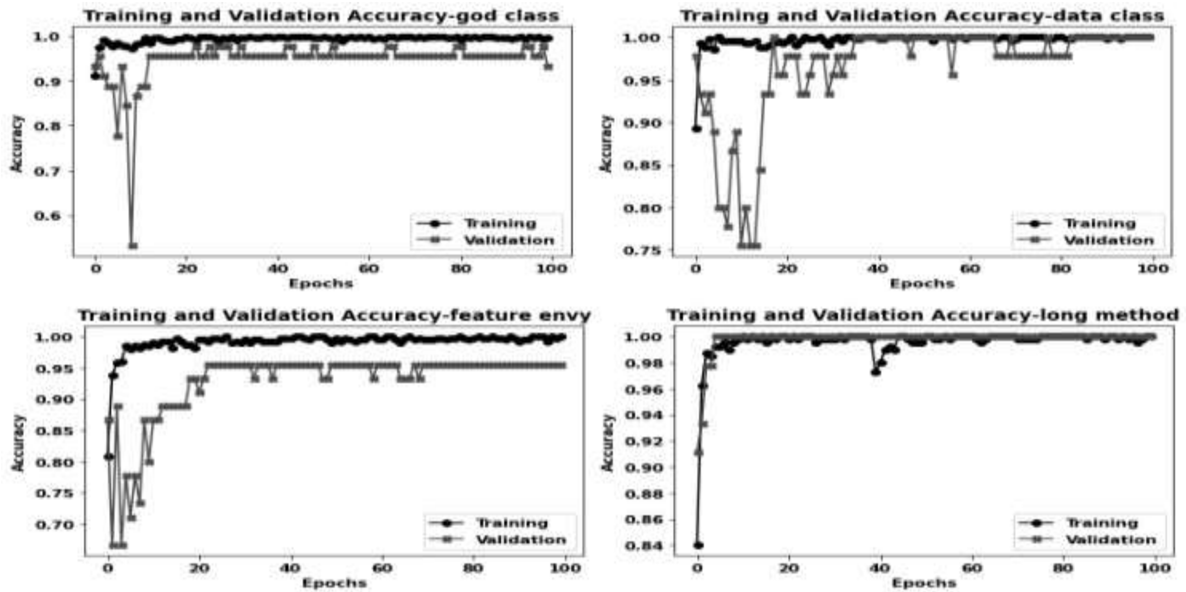


Figure 7.7 Training and Validation Accuracy over balanced datasets

Figures 7.8 and 7.9 show the training and validation loss of the model on the original and balanced datasets. The vertical axis presents the loss of the model, and the horizontal axis illustrates the number of epochs. The loss indicates how bad a model prediction was.
Figure 7.8 shows the loss values of the model on the original datasets. From the Figure, the model loss is 0.036 for the God Class dataset, 0.005 for the Data Class dataset, 0.041 for the Feature Envy dataset, and 0.021 for the Long Method dataset at the 100th epoch.

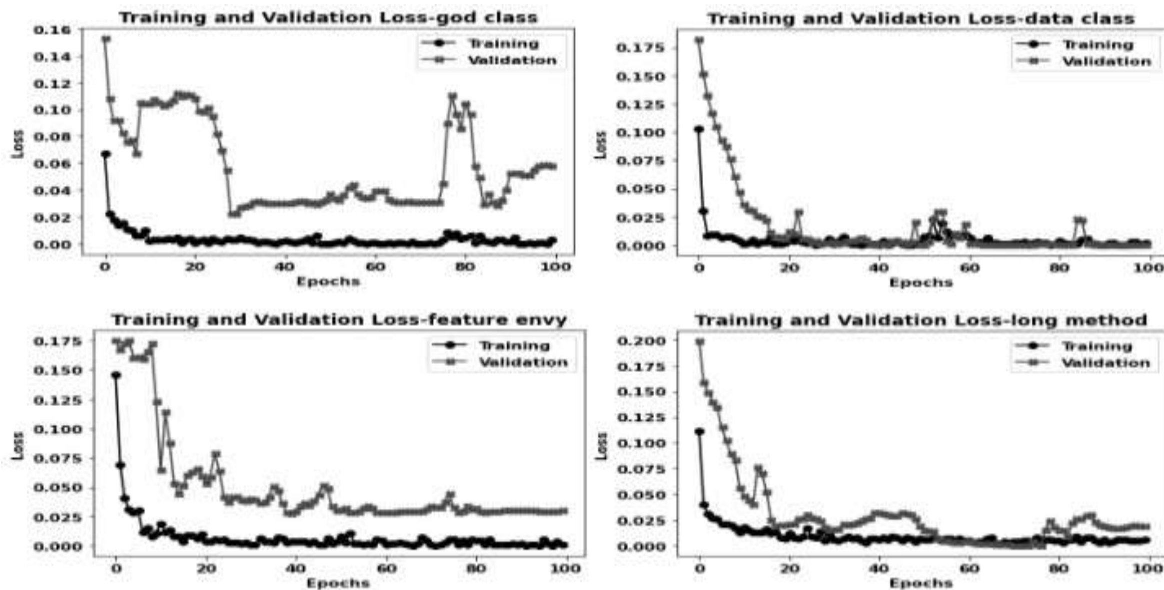


Figure 7.8 Training and validation loss over original datasets

Figure 7.9 shows the loss values of the model on the balanced datasets. From the Figure, the model loss is 0.033 for the God Class dataset, 0.013 for the Data Class dataset, 0.018 for the Feature Envy dataset, and 0.000 for the Long Method dataset at the 100th epoch.
As shown in the Figures, the accuracy of training and validation increases and the loss decreases with increasing epochs. Regarding the high accuracy and low loss obtained by the proposed model, we note that the model is well-trained and validated.

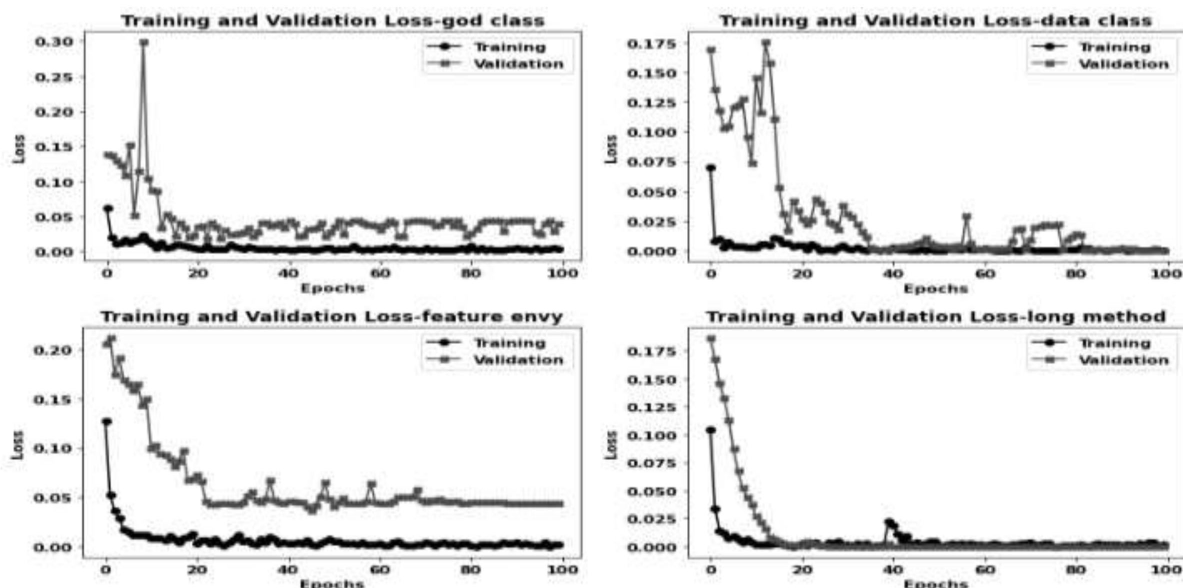


Figure 7.9 Training and validation loss over balanced datasets

Table 7.9 shows the comparison results of our method with some previous studies based on accuracy. The best values are indicated in bold and "-" indicates that the approaches that did not provide results for performance measures in a particular data set. According to Table, some of the results in the previous studies are better than ours. However, in most cases, our method outperforms the other state-of-the-art methods and provides better predictive performance.

Table 7.9 Comparison of the proposed method with other existing methods based on the accuracy

Methods	**Datasets**			
	God class	**Data class**	**Feature envy**	**Long method**
RF[4]	0.96	0.98	0.96	0.99
NB[4]	0.97	0.97	0.91	0.97
DT[27]	-	-	0.97	-
RF[27]	-	**0.99**	-	0.95
K-NN[108]	0.97	0.97	0.91	0.97
NB[108]	0.96	0.84	0.92	0.95
MLP[108]	0.97	0.97	0.95	0.96
DT[108]	0.97	0.98	**0.98**	0.98
RF[108]	0.97	0.98	0.97	0.99
LR[108]	0.97	0.97	0.97	0.99
NB[118]	0.96	-	0.91	0.97
MLP[118]	0.97	-	0.92	0.99
DT[118]	**0.98**	-	0.95	0.97
RF[119]	0.76	0.81	0.66	0.60
NB[119]	0.74	0.66	0.76	0.74
SVM[119]	0.66	0.66	0.60	0.66
RF[120]	0.69	0.70	0.71	0.68
NB[120]	0.82	0.75	0.83	0.81
SVM[120]	0.74	0.83	0.83	0.81
K-NN[120]	0.80	0.82	0.82	0.81
Our model (CNN) - Original Datasets	0.96	**0.99**	0.95	0.98
Our model (CNN with SMOTE) - Balanced Datasets	0.96	0.98	**0.98**	**1.00**

In summary, this study aimed to present a method based on CNN with the Oversampling method (SMOTE) to detect code smells. We compared the results obtained by the proposed method based on the original and balanced datasets to investigate the impact of Oversampling methods on improving the accuracy of ML techniques. Additionally, the proposed method's results were compared with those presented in previous studies. After comparing the results obtained by the proposed model on the original datasets with results obtained by the proposed model on the balanced datasets, as shown in the Tables and Figures, we note that the model got good scores on the balanced datasets and the results improved further due to balancing, which indicated that the combination of CNN with the Oversampling method (SMOTE) has a positive effect on the performance of code smells detection regarding datasets with imbalanced class distributions. Furthermore, data sampling methods play an essential role in improving the accuracy of the ML models in code smells detection. Regarding the evaluation of the results obtained from our proposed method and their comparison with some results of other studies, we conclude that our model is promising in code smell detection and outperforms other models in the previous studies.

7.3 Bi-LSTM and GRU with Under and Oversampling Methods in Code Smells Detection

In this sub-section, we discuss the findings of the seventh study, the objective was to present a method based on RNN models (Bi-LSTM and GRU) with Under and Oversampling methods (Random Oversampling and Tomek Links) to detect four code smells (God class, data class, feature envy, and long method). The experiments have been conducted based on benchmark datasets obtained from the Qualitas Corpus Systems. The experimental results were evaluated and compared based on various performance measures (accuracy, precision, recall, f-measure, MCC, AUC, AUCPR, MSE).
The performance of the prediction models is reported in Tables 7.10 to 7.18 and Figures 7.10 to 7.18, appendix 4 (Figures 1 to 12).
Table 7.10 presents the results of Bi-LSTM and GRU models on the original datasets in terms of accuracy, precision, recall, F-Measure, MCC, AUC, AUCPR and MSE. We notice that the accuracy values of the Bi-LSTM model range from 0.95 to 0.98, the precision values range from 0.93 to 1.00, the recall values range from 0.83 to 0.96, the F-Measure values range from 0.90 to 0.96, the MCC values range from 0.88 to 0.94, the AUC values range from 0.97 to 0.99, the AUCPR values range from 0.95 to 0.99, and the MSE values range from 0.023 to 0.044 across all datasets. The accuracy values of the GRU model range from 0.93 to 0.98, the precision values range from 0.86 to 0.97, the recall values range from 0.86 to 0.96, the F-Measure values range from 0.89 to 0.96, the MCC values range from 0.84 to 0.94, the AUC values range from 0.95 to 0.99, the AUCPR values range from 0.89 to 0.99, and the MSE values range from 0.020 to 0.065 across all datasets.

Table 7.10 Evaluation results for the original datasets

Bi-LSTM Model								
Datasets	**Performance Measures**							
	Accuracy	**Precision**	**Recall**	**F- measure**	**MCC**	**AUC**	**AUCPR**	**MSE**
God Class	0.95	0.97	0.92	0.94	0.90	0.99	0.99	0.035
Data Class	0.95	1.00	0.83	0.90	0.88	0.99	0.99	0.037
Feature envy	0.95	0.93	0.93	0.93	0.89	0.97	0.95	0.044
Long method	0.98	0.96	0.96	0.96	0.94	0.99	0.99	0.023
Averages	**0.95**	**0.96**	**0.91**	**0.93**	**0.90**	**0.98**	**0.98**	**0.034**

GRU Model								
Datasets	Performance Measures							
	Accuracy	Precision	Recall	F- measure	MCC	AUC	AUCPR	MSE
God Class	0.93	0.97	0.86	0.91	0.85	0.97	0.97	0.063
Data Class	0.96	0.92	0.96	0.94	0.91	0.99	0.99	0.026
Feature envy	0.93	0.86	0.93	0.89	0.84	0.95	0.89	0.065
Long method	0.98	0.96	0.96	0.96	0.94	0.99	0.99	0.020
Averages	**0.95**	**0.92**	**0.92**	**0.92**	**0.88**	**0.97**	**0.96**	**0.043**

Table 7.11 presents the results of Bi-LSTM and GRU Models on the balanced datasets using Random Oversampling regarding accuracy, precision, recall, F-Measure, MCC, AUC, AUCPR and MSE. We notice that the accuracy values of the Bi-LSTM model range from 0.96 to 1.00, the precision values range from 0.94 to 1.00, the recall values range from 0.98 to 1.00, the F-Measure values range from 0.97 to 1.00, the MCC values range from 0.92 to 1.00, the AUC values range from 0.97 to 1.00, the AUCPR values range from 0.96 to 1.00, and the MSE values range from 0.005 to 0.037 across all datasets. The accuracy values of the GRU model range from 0.96 to 1.00, the precision values range from 0.95 to 1.00, the recall value range from 0.98 to 1.00, the F-Measure values range from 0.97 to 1.00, the MCC values range from 0.92 to 1.00, the AUC values range from 0.96 to 1.00, the AUCPR values range from 0.93 to 1.00, and the MSE values range from 0.002 to 0.033 across all datasets.

Table 7.11 Evaluation results for the balanced datasets - Random Oversampling

Bi-LSTM Model								
Datasets	Performance Measures							
	Accuracy	Precision	Recall	F- measure	MCC	AUC	AUCPR	MSE
God Class	0.96	0.95	0.98	0.97	0.92	0.98	0.98	0.035
Data Class	0.99	0.98	1.00	0.99	0.98	1.00	1.00	0.006
Feature envy	0.96	0.94	1.00	0.97	0.92	0.97	0.96	0.037
Long method	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.005
Averages	**0.97**	**0.96**	**0.99**	**0.98**	**0.95**	**0.98**	**0.98**	**0.020**
GRU Model								
Datasets	Performance Measures							
	Accuracy	Precision	Recall	F- measure	MCC	AUC	AUCPR	MSE
God Class	0.96	0.95	0.98	0.97	0.92	0.96	0.93	0.033
Data Class	0.98	0.98	0.98	0.98	0.96	0.99	0.99	0.023
Feature envy	0.97	0.95	1.00	0.98	0.94	0.97	0.95	0.032
Long method	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.002
Averages	**0.97**	**0.97**	**0.99**	**0.98**	**0.95**	**0.98**	**0.96**	**0.022**

Table 7.12 presents the results of Bi-LSTM and GRU Models on the balanced datasets using Tomek links regarding accuracy, precision, recall, F-Measure, MCC, AUC, AUCPR and MSE. We notice that the accuracy values of the Bi-LSTM model range from 0.95 to 0.99, the precision values range from 0.85 to 1.00, the recall values range from 0.87 to 1.00, the F-Measure values range from 0.92 to 0.98, the MCC values range from 0.88 to 0.97, the AUC values range from 0.97 to 0.99, the AUCPR values range from 0.92 to 0.98, and the MSE values range from 0.013 to 0.044 across all datasets. The accuracy values of the GRU model range

from 0.96 to 0.99, the precision values range from 0.94 to 1.00, the recall values range from 0.87 to 1.00, the F-Measure values range from 0.93 to 0.98, the MCC values range from 0.90 to 0.97, the AUC values range from 0.98 to 0.99, the AUCPR values range from 0.97 to 0.99, and the MSE values range from 0.018 to 0.038 across all datasets.

Table 7.12 Evaluation results for the balanced datasets - Tomek links

Bi-LSTM Model								
Datasets	**Performance Measures**							
	Accuracy	**Precision**	**Recall**	**F- measure**	**MCC**	**AUC**	**AUCPR**	**MSE**
God Class	0.96	1.00	0.87	0.93	0.90	0.98	0.97	0.037
Data Class	0.95	0.85	1.00	0.92	0.88	0.97	0.92	0.044
Feature envy	0.98	0.97	0.97	0.97	0.94	0.99	0.98	0.020
Long method	0.99	0.97	1.00	0.98	0.97	0.98	0.97	0.013
Averages	**0.97**	**0.94**	**0.96**	**0.95**	**0.92**	**0.98**	**0.96**	**0.028**
GRU Model								
Datasets	**Performance Measures**							
	Accuracy	**Precision**	**Recall**	**F- measure**	**MCC**	**AUC**	**AUCPR**	**MSE**
God Class	0.96	1.00	0.87	0.93	0.90	0.98	0.97	0.038
Data Class	0.99	0.96	1.00	0.98	0.97	0.99	0.99	0.018
Feature envy	0.99	0.97	1.00	0.98	0.97	0.99	0.99	0.021
Long method	0.98	0.94	1.00	0.97	0.94	0.99	0.99	0.025
Averages	**0.98**	**0.96**	**0.96**	**0.96**	**0.94**	**0.98**	**0.98**	**0.025**

Figures 7.10 to 7.13 show the training and validation accuracy and training and validation loss of the models on the original datasets.

Figures 7.10 and 7.11 show the training and validation accuracy of the models on the original datasets. The vertical axis presents the accuracy of the models, and the horizontal axis illustrates the number of epochs. Accuracy is the fraction of predictions that our models predicted right.

Figure 7.10 shows the accuracy values of the Bi-LSTM model. From the Figure, the model learned 95% accuracy for God Class, 95% accuracy for Data Class, 95% accuracy for Feature envy and 98% accuracy for Long method at the 100th epoch.

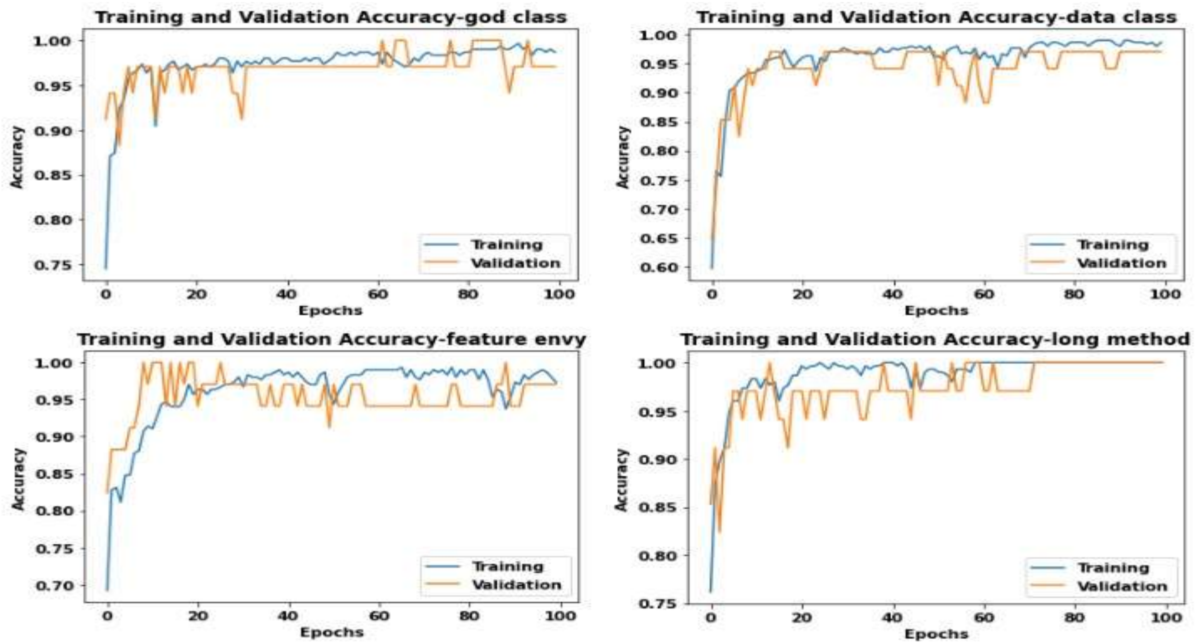


Figure 7.10 Training and Validation Accuracy on the original datasets using Bi-LSTM Model

Figure 7.11 shows the accuracy values of the GRU model. From the Figure, the model learned 93% accuracy for God Class, 96% accuracy for Data Class, 93% accuracy for Feature envy and 98% accuracy for Long method at the 100th epoch.

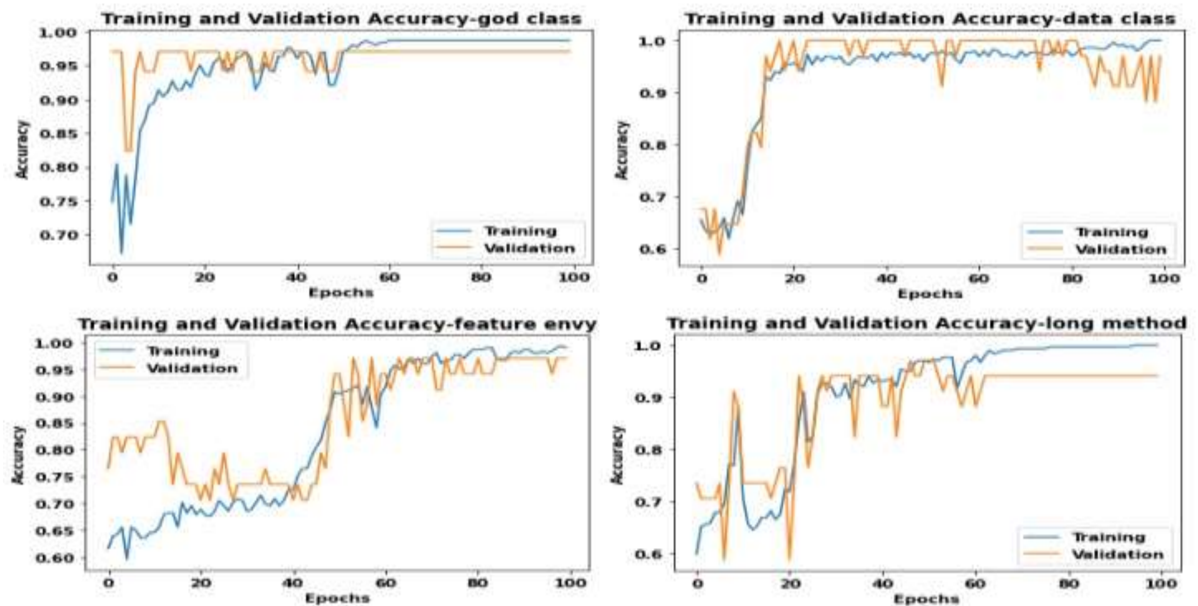


Figure 7.11 Training and Validation Accuracy on the original datasets using GRU Model

Figures 7.12 and 7.13 show the training and validation loss of the models on the original datasets. The vertical axis presents the loss of the models, and the horizontal axis illustrates the number of epochs. The loss indicates how bad a model's prediction was.

Figure 7.12 shows the loss values of the Bi-LSTM model. From the Figure, the model loss is 0.035 for God Class, 0.037 for Data Class, 0.044 for Feature envy and 0.023 for the long method at the 100th epoch.

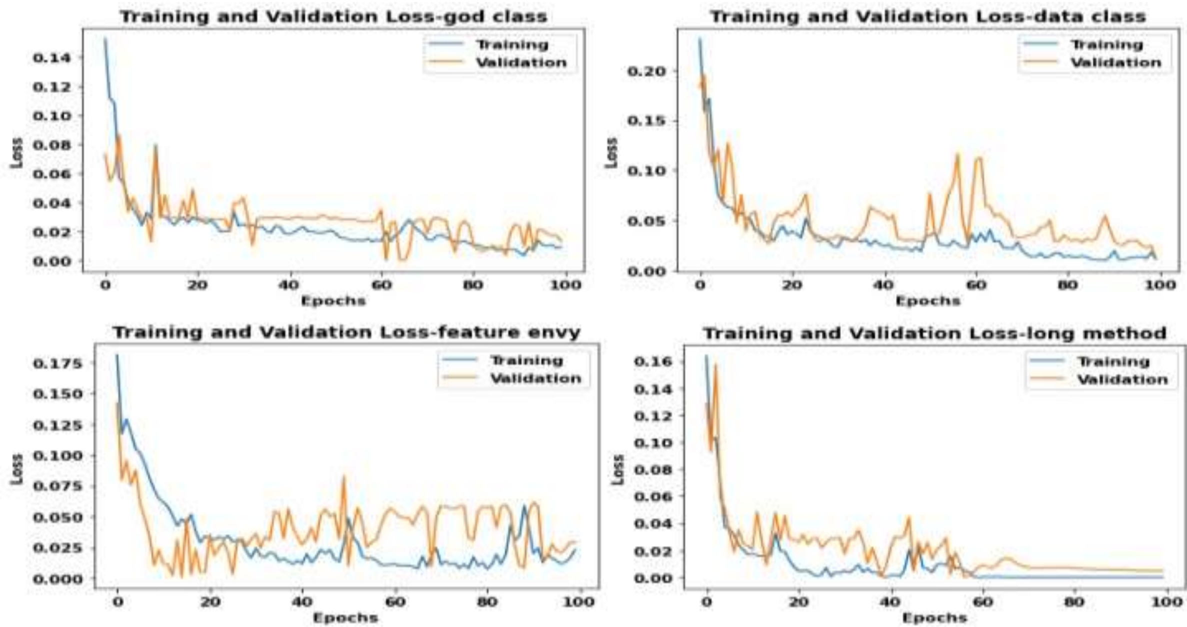


Figure 7.12 Training and Validation Loss on the original datasets using Bi-LSTM Model

Figure 7.13 shows the loss values of the GRU model. From the Figure, the model loss is 0.063 for God Class, 0.026 for Data Class, 0.065 for Feature envy and 0.020 for the long method at the 100th epoch. Further in appendix 4, Figures 1 to 8 show both models' training and validation (accuracy and loss) on the balanced datasets.
As shown in the Figures, the accuracy of training and validation increases, and the loss decreases with increasing epochs. Regarding the high accuracy and low loss obtained by the proposed models, we note that both models are well-trained and validated. Additionally, we note that the models are approximately perfectly fitting, there is no overfitting or underfitting.

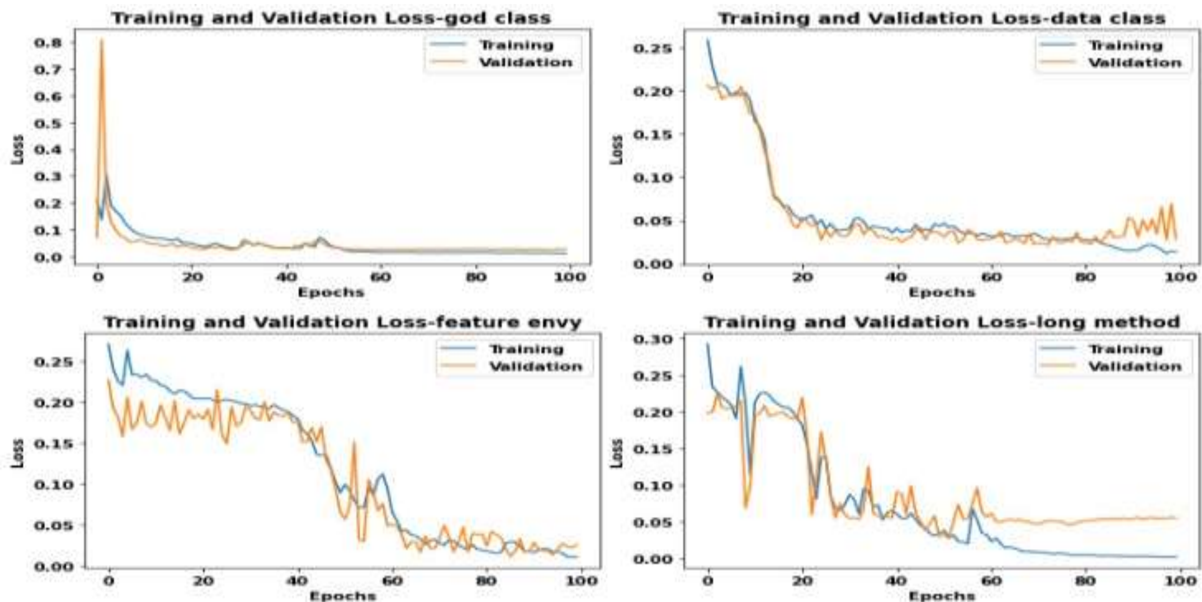


Figure 7.13 Training and Validation Loss on the original datasets using GRU Model

Figures 7.14 and 7.15 show the ROC curves of the models on the original datasets. The vertical axis presents the actual positive rate of the models, and the horizontal axis illustrates the false

positive rate. The AUC is a sign of the performance of the model. The larger AUC is, the better the model performance will be. Based on the Figures, the values are encouraging and indicate our proposed models' efficiency in detecting code smells.

Figure 7.14 shows the AUC values of the Bi-LSTM model. From the Figure, the AUC values are 99% on God Class,99% on Data Class, 95% on Feature envy and 99% on the Long method.

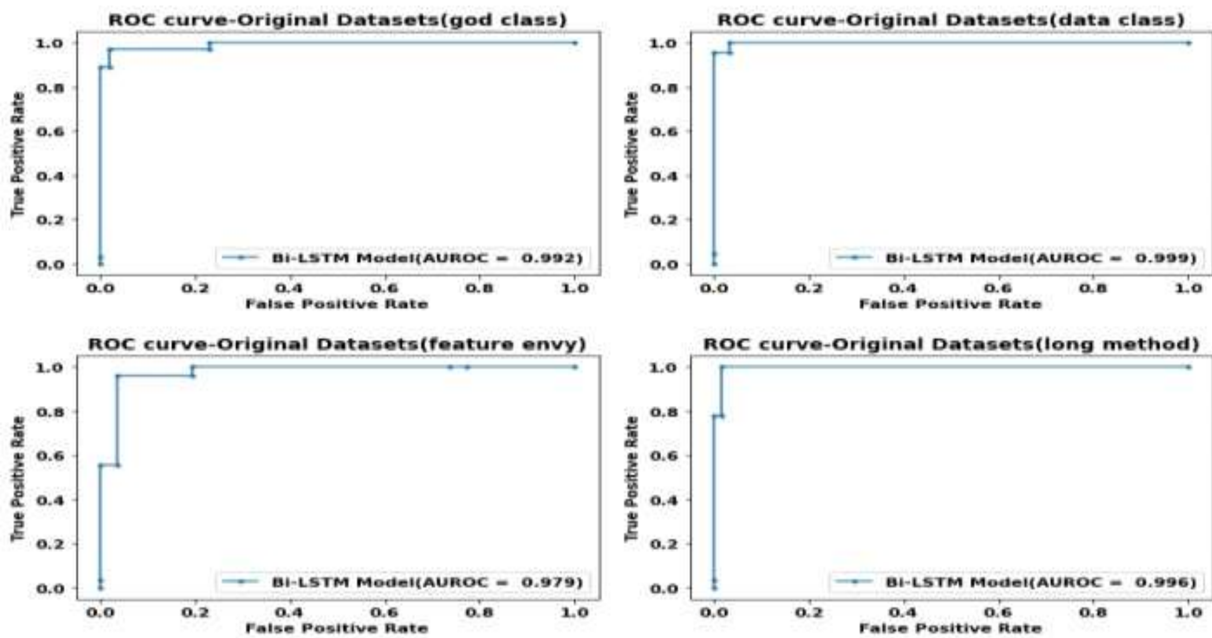


Figure 7.14 ROC curves for the original datasets - Bi-LSTM Model

Figure 7.15 shows the AUC values of the GRU model. From the Figure, the AUC values are 97% on God Class,99% on Data Class, 89% on Feature envy and 99% on the Long method. Further in appendix 4, Figures 9 to 12 show the ROC curves for both models on the balanced datasets.

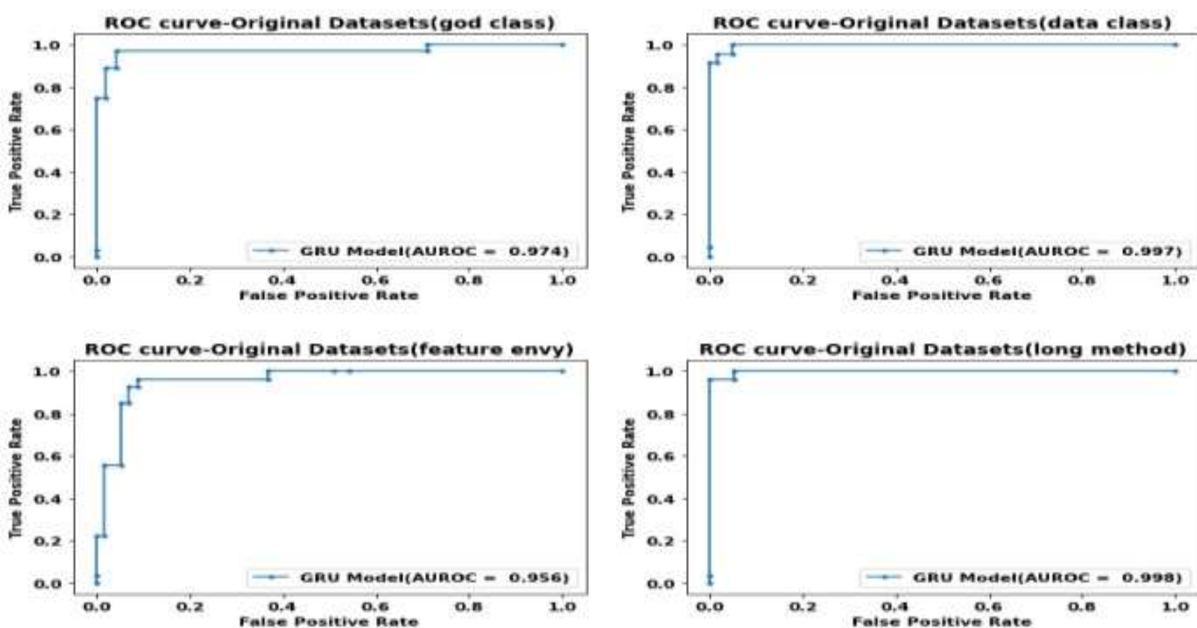


Figure 7.15 ROC curves for the original datasets - GRU Model

We used Boxplots to aggregate the achieved results to get a more accurate overview of the quality of the results. Figure 7.16 shows the Box plots for the original datasets' performance measures. For the Bi-LSTM model, the highest accuracy is 98% on the Long method dataset and the lowest accuracy is 95% on the God Class, Data Class and Feature envy datasets, the highest precision is 100% on the Data Class dataset and the lowest precision is 93% on the Feature envy dataset, the highest recall is 96% on the Long method dataset and the lowest recall is 83% on the Data Class dataset, the highest f-measure is 96% on the Long method dataset and the lowest f-measure is 90% on the Data Class dataset, the highest MCC is 94% on the Long method dataset and the lowest MCC is 88% on the Data Class dataset, the highest AUC is 99% on the God Class, Data Class and Long method datasets and the lowest AUC is 97% on the Feature envy dataset, the highest AUCPR is 99% on the God Class, Data Class and Long method datasets and the lowest AUCPR is 95% on the Feature envy dataset.
In contrast, For the GRU model, the highest accuracy is 98% on the Long method dataset and the lowest accuracy is 93% on the God Class and Feature envy datasets, the highest precision is 97% on the God Class dataset and the lowest precision is 86% on the Feature envy dataset, the highest recall is 96% on the Data Class and Long method datasets and the lowest recall is 86% on the God Class dataset, the highest f-measure is 96% on the Long method dataset and the lowest f-measure is 89% on the Feature envy dataset, the highest MCC is 94% on the Long method dataset and the lowest MCC is 84% on the Feature envy dataset, the highest AUC is 99% on the Data Class and Long method datasets and the lowest AUC is 95% on the Feature envy dataset, the highest AUCPR is 99% on the Data Class and Long method datasets and the lowest AUCPR is 89% on the Feature envy dataset.

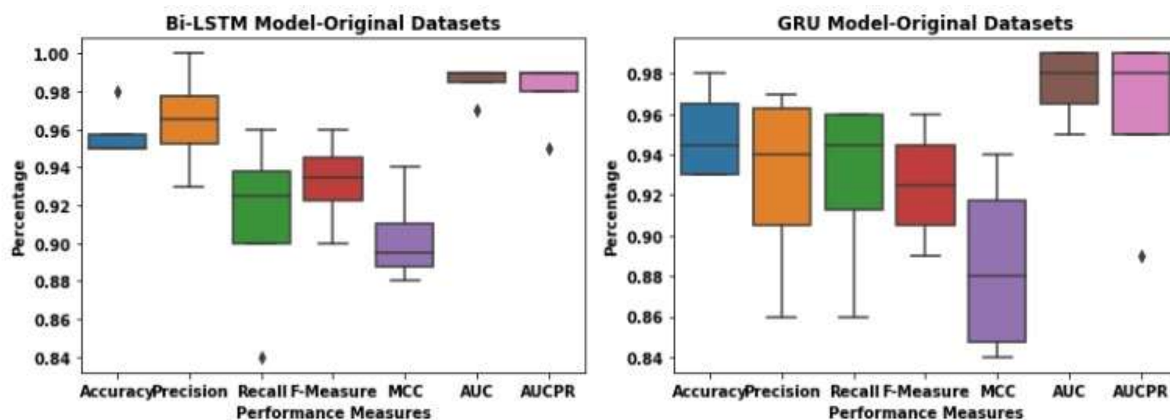


Figure 7.16 Boxplots representing performance measures obtained by models on the original datasets

Figure 7.17 shows the Box plots for the performance measures on the balanced datasets using Random Oversampling. For the Bi-LSTM model with Random Oversampling, the highest accuracy is 100% on the Long method dataset and the lowest accuracy is 96% on the God Class and Feature envy datasets, the highest precision is 100% on the Long method dataset and the lowest precision is 94% on the Feature envy dataset, the highest recall is 100% on the Data Class, Feature envy and Long method datasets and the lowest recall is 98% on the God Class dataset, the highest f-measure is 100% on the Long method dataset and the lowest f-measure is 97% on the God Class and Feature envy datasets, the highest MCC is 100% on the Long method dataset and the lowest MCC is 92% on the God Class and Feature envy datasets, the highest AUC is 100% on the Data Class and Long method datasets and the lowest AUC is 97%

on the Feature envy dataset, the highest AUCPR is 100% on the Data Class and Long method datasets and the lowest AUCPR is 96% on the Feature envy dataset.
In contrast, For the GRU model with Random Oversampling, the highest accuracy is 100% on the Long method dataset and the lowest accuracy is 96% on the God Class dataset, the highest precision is 100% on the Long method dataset and the lowest precision is 95% on the God Class and Feature envy datasets, the highest recall is 100% on the Feature envy and Long method datasets and the lowest recall is 98% on the God Class and Data Class datasets, the highest f-measure is 100% on the Long method dataset and the lowest f-measure is 97% on the God Class dataset, the highest MCC is 100% on the Long method dataset and the lowest MCC is 92% on the God Class dataset, the highest AUC is 100% on the Long method dataset and the lowest AUC is 96% on the God Class dataset, the highest AUCPR is 100% on the Long method dataset and the lowest AUCPR is 93% on the God Class dataset.

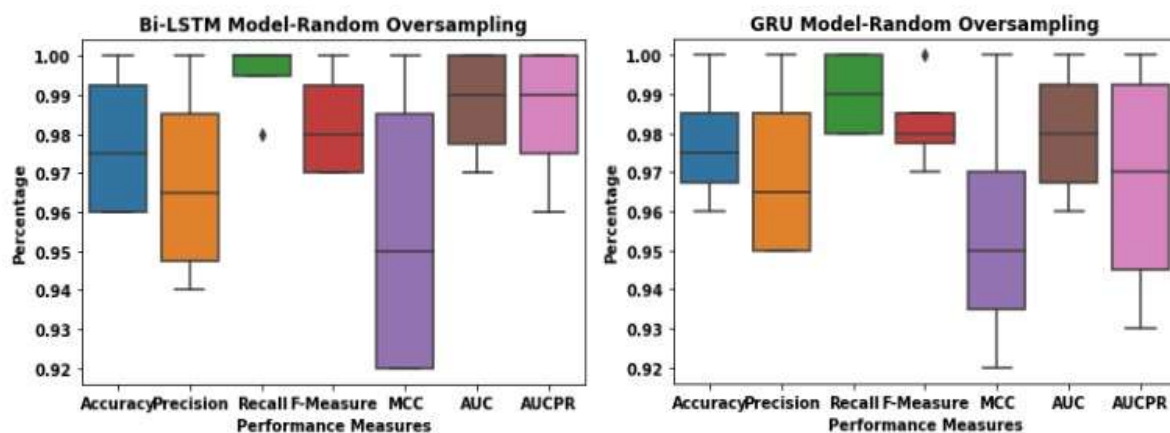


Figure 7.17 Boxplots representing performance measures obtained by models on the balanced datasets- Random Oversampling

Figure 7.18 shows the Box plots for the performance measures on the balanced datasets using Tomek links. For the Bi-LSTM model with Tomek links, the highest accuracy is 99% on the Long method dataset and the lowest accuracy is 95% on the Data Class dataset, the highest precision is 100% on the God Class dataset and the lowest precision is 85% on the Data Class dataset, the highest recall is 100% on the Data Class and Long method datasets and the lowest recall is 87% on the God Class dataset, the highest f-measure is 98% on the Long method dataset and the lowest f-measure is 92% on the Data Class dataset, the highest MCC is 97% on the Long method dataset and the lowest MCC is 88% on the Data Class dataset, the highest AUC is 99% on the Feature envy dataset and the lowest AUC is 97% on the Data Class dataset, the highest AUCPR is 98% on the Feature envy dataset and the lowest AUCPR is 92% on the Data Class dataset.
In contrast, For the GRU model with Tomek links, the highest accuracy is 99% on the Data Class and Feature envy datasets and the lowest accuracy is 96% on the God Class dataset, the highest precision is 100% on the God Class dataset and the lowest precision is 94% on the Long method dataset, the highest recall is 100% on the Data Class, Feature envy and Long method datasets and the lowest recall is 87% on the God Class dataset, the highest f-measure is 98% on the Data Class and Feature envy datasets and the lowest f-measure is 93% on the God Class dataset, the highest MCC is 97% on the Data Class and Feature envy datasets and the lowest MCC is 90% on the God Class dataset, the highest AUC is 99% on the Data Class, Feature envy and Long method datasets and the lowest AUC is 98% on the God Class dataset,

the highest AUCPR is 99% on the Data Class, Feature envy and Long method datasets and the lowest AUCPR is 97% on the God Class dataset.

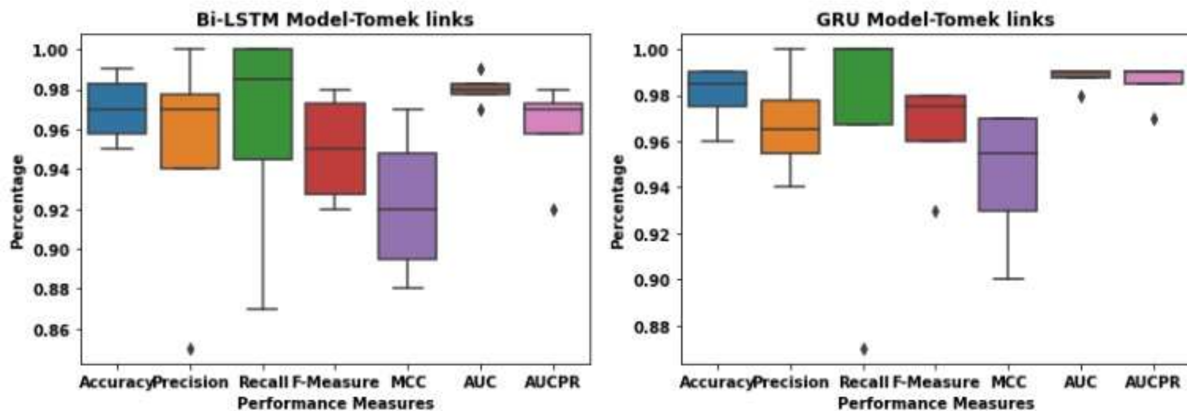


Figure 7.18 Boxplots representing performance measures obtained by models on the balanced datasets- Tomek links

Table 7.13 presents the statistical analysis results (paired t-test) of proposed models on the original and balanced datasets (using Random Oversampling) in terms of mean, Standard Deviation (STD), min, max and P value. We notice that the mean values of the Bi-LSTM model are 0.95 on the original datasets and 0.97 on the balanced datasets. The mean values of the GRU model are 0.95 on the original datasets and 0.97 on the balanced datasets. The STD values of the Bi-LSTM model are 0.01 on the original datasets and 0.02 on the balanced datasets, while the STD values of the GRU model are 0.02 on the original datasets and 0.01 on the balanced datasets. The Min values of the Bi-LSTM model are 0.95 on the original datasets and 0.96 on the balanced datasets, while the Min values of the GRU model are 0.93 on the original datasets and 0.96 on the balanced datasets. The Max values of the Bi-LSTM model are 0.98 on the original datasets and 1.00 on the balanced datasets, while the Max values of the GRU model are 0.98 on the original datasets and 1.00 on the balanced datasets. The P value of the Bi-LSTM model is 0.06 for the original and balanced datasets, while the P value of the GRU model is 0.01 for the original and balanced datasets. Based on the P value of the GRU model on the original and balanced data sets, we note that the P value is less than 0.05, indicating a difference between the results of the models on the original and balanced data sets.

Table 7.13 Comparison of the proposed models in terms of accuracy using paired t-test- based on the original and balanced datasets (using Random Oversampling)

Paired t-test	Bi-LSTM Model		GRU Model	
	Original Datasets	Balanced Datasets	Original Datasets	Balanced Datasets
Mean	0.95	0.97	0.95	0.97
STD	0.01	0.02	0.02	0.01
Min	0.95	0.96	0.93	0.96
Max	0.98	1.00	0.98	1.00
P value	0.06		0.01	

Table 7.14 presents the statistical analysis results (paired t-test) of proposed models on the original and balanced datasets (using Tomek Links) in terms of mean, Standard Deviation (STD), min, max and P value. We notice that the mean values of the Bi-LSTM model are 0.95 on the original datasets and 0.97 on the balanced datasets. The mean values of the GRU model are 0.95 on the original datasets and 0.98 on the balanced datasets. The STD values of the Bi-

LSTM model are 0.01 on the original datasets and 0.01 on the balanced datasets, while the STD values of the GRU model are 0.02 on the original datasets and 0.01 on the balanced datasets. The Min values of the Bi-LSTM model are 0.95 on the original datasets and 0.95 on the balanced datasets, while the Min values of the GRU model are 0.93 on the original datasets and 0.96 on the balanced datasets. The Max values of the Bi-LSTM model are 0.98 on the original datasets and 0.99 on the balanced datasets, while the Max values of the GRU model are 0.98 on the original datasets and 0.99 on the balanced datasets. The P value of the Bi-LSTM model is 0.14 for the original and balanced datasets, while the P value of the GRU model is 0.09 for the original and balanced datasets. Based on the P value of both models on the original and balanced data sets, we note that the P value is greater than 0.05, indicating no difference between the results of the models on the original and balanced data sets.

Table 7.14 Comparison of the proposed models in terms of accuracy using paired t-test- based on the original and balanced datasets (using Tomek Links)

Paired t-test	**Bi-LSTM Model**		**GRU Model**	
	Original Datasets	**Balanced Datasets**	**Original Datasets**	**Balanced Datasets**
Mean	0.95	0.97	0.95	0.98
STD	0.01	0.01	0.02	0.01
Min	0.95	0.95	0.93	0.96
Max	0.98	0.99	0.98	0.99
P value	0.14		0.09	

The results presented by our models and previous studies' results are reported in Tables 7.15 to 7.18. Tables 7.15 and 7.16 show the comparison results of our method with some previous studies based on some performance measures, namely accuracy and AUC. Table 7.15 shows the results based on accuracy; Table 7.16 shows the results based on AUC. The best values are indicated in bold in the Tables and "- "indicates that the approaches that did not provide results for performance measures in a particular data set. According to Tables 7.15 and 7.16, some of the results in the previous studies are better than ours. However, in most cases, our method outperforms the other state-of-the-art approaches and provides better predictive performance.

Table 7.15 Comparison of the proposed models with other existing approaches based on the accuracy

Approaches	**Datasets**				**Averages**
	God class	**Data class**	**Feature envy**	**Long method**	
RF[4]	0.96	0.98	0.96	0.99	0.97
NB[4]	**0.97**	0.97	0.91	0.97	0.95
DT[27]	-	-	0.97	-	0.97
RF[27]	-	**0.99**	-	0.95	0.97
K-NN[108]	**0.97**	0.97	0.91	0.97	0.95
NB[108]	0.96	0.84	0.92	0.95	0.91
MLP[108]	**0.97**	0.97	0.95	0.96	0.96
DT[108]	**0.97**	0.98	0.98	0.98	0.97
RF[108]	**0.97**	0.98	0.97	0.99	0.97
LR[108]	**0.97**	0.97	0.97	0.99	0.97
RF[120]	0.69	0.70	0.71	0.68	0.69
NB[120]	0.82	0.75	0.83	0.81	0.80
SVM[120]	0.74	0.83	0.83	0.81	0.80
K-NN[120]	0.80	0.82	0.82	0.81	0.81
Our Bi-LSTM model_Balanced Datasets (Random Oversampling)	0.96	**0.99**	0.96	**1.00**	0.97
Our GRU model_Balanced Datasets (Random Oversampling)	0.96	0.98	0.97	**1.00**	0.97

Our Bi-LSTM model_Balanced Datasets (Tomek links)	0.96	0.95	0.98	0.99	0.97
Our GRU model_Balanced Datasets (Tomek links)	0.96	**0.99**	**0.99**	0.98	**0.98**

Table 7.16 Comparison of the proposed models with other existing approaches based on AUC

Approaches	**Datasets**				**Averages**
	God class	**Data class**	**Feature envy**	**Long method**	
DL[29]	-	-	0.84	0.79	0.81
RF[120]	0.59	0.65	0.59	0.52	0.58
NB[120]	0.88	0.85	0.86	0.86	0.86
SVM[120]	0.65	0.88	0.82	0.66	0.75
K-NN[120]	0.83	0.86	0.83	0.86	0.84
Our Bi-LSTM model_Balanced Datasets (Random Oversampling)	**0.98**	**1.00**	0.97	**1.00**	**0.98**
Our GRU model_Balanced Datasets (Random Oversampling)	0.96	0.99	0.97	**1.00**	**0.98**
Our Bi-LSTM model_Balanced Datasets (Tomek links)	**0.98**	0.97	**0.99**	0.98	**0.98**
Our GRU model_Balanced Datasets (Tomek links)	**0.98**	0.99	**0.99**	0.99	**0.98**

Table 7.17 presents the statistical analysis results (paired t-test) for the proposed models based on Random Oversampling and existing approaches in terms of mean, Standard Deviation (STD), min, max and P value. We notice that the mean value of existing approaches is 0.90, while the mean value of the Bi-LSTM Model is 0.97 and the mean value of the GRU Model is 0.97. The STD value of existing approaches is 0.00, while the STD value of the Bi-LSTM Model is 0.02 and the STD value of the GRU Model is 0.01. The Min value of existing approaches is 0.89, while the Min value of the Bi-LSTM Model is 0.96 and the Min value of the GRU Model is 0.96. The Max value of existing approaches is 0.91, while the Max value of the Bi-LSTM Model is 1.00 and the Max value of the GRU Model is 0.99. The P value for existing approaches and Bi-LSTM Model is 0.00, while the P value for existing approaches and GRU Model is 0.00. Based on the P value of both models based on Random Oversampling and existing approaches, we note that the P value is less than 0.05, indicating a difference between the existing approaches' and our proposed models' results.

Table 7.17 Comparison of the proposed models with other existing approaches in terms of accuracy averages using paired t-test- based on Random Oversampling

Paired t-test	**Existing Approaches**	**Bi-LSTM Model**	**Existing Approaches**	**GRU Model**
Mean	0.90	0.97	0.90	0.97
STD	0.00	0.02	0.00	0.01
Min	0.89	0.96	0.89	0.95
Max	0.91	1.00	0.91	0.99
P value	0.00		0.00	

Table 7.18 presents the statistical analysis results (paired t-test) for the proposed models based on Tomek Links and existing approaches in terms of mean, Standard Deviation (STD), min, max and P value. We notice that the mean value of existing approaches is 0.90, while the mean value of the Bi-LSTM Model is 0.97 and the mean value of the GRU Model is 0.98. The STD value of existing approaches is 0.00, while the STD value of the Bi-LSTM Model is 0.01 and the STD value of the GRU Model is 0.01. The Min value of existing approaches is 0.89, while the Min value of the Bi-LSTM Model is 0.96 and the Min value of the GRU Model is 0.96. The Max value of existing approaches is 0.91, while the Max value of the Bi-LSTM Model is

1.00 and the Max value of the GRU Model is 0.99. The P value for existing approaches and Bi-LSTM Model is 0.00, while the P value for existing approaches and GRU Model is 0.00. Based on the P value of both models based on Tomek Links and existing approaches, we note that the P value is less than 0.05, indicating a difference between the existing approaches' and our proposed models' results.

Table 7.18 Comparison of the proposed models with other existing approaches in terms of accuracy averages using paired t-test- based on Tomek Links

Paired t-test	Existing Approaches	Bi-LSTM Model	Existing Approaches	GRU Model
Mean	0.90	0.97	0.90	0.98
STD	0.00	0.01	0.00	0.01
Min	0.89	0.96	0.89	0.96
Max	0.91	1.00	0.91	0.99
P value	0.00		0.00	

In summary, this study aimed to present a method based on RNN models (Bi-LSTM and GRU) with Under and Oversampling methods (Random Oversampling and Tomek Links) to detect code smells. We compared the results obtained by the proposed method based on the original and balanced datasets to investigate the impact of Under and Oversampling methods on improving the accuracy of ML techniques. Additionally, the proposed method's results were compared with those presented in previous studies. After comparing the results obtained by the proposed models on the original datasets with results obtained by the proposed models on the balanced datasets, as shown in the Tables and Figures, we note that the models got good scores on the balanced datasets and the results improved further due to balancing, which indicated that the combination of (Bi-LSTM and GRU) with Under and Oversampling methods (Random Oversampling and Tomek Links) has positive effect on the performance of code smells detection regarding datasets with imbalanced class distributions. Furthermore, data sampling methods play an essential role in improving the accuracy of ML models in code smells detection. Regarding the evaluation of the results obtained from our proposed method and their comparison with some results of other studies, we conclude that our models are promising in code smell detection and outperform other models in the previous studies.

7.4 Summary

In this chapter, we presented the experimental results and discussion of code smells detection. The experimental results have been compared and evaluated based on several standard performance measures. We compared experimental results based on the original and balanced datasets and compared our results with current state-of-the-art results for code smells detection. The results showed that our proposed methods significantly outperform current state-of-the-art methods for code smells detection. We concluded that the combined data-balancing methods with ML techniques significantly enhance the accuracy of code smells detection. We observe that the incorporation of appropriate data-balancing methods and ML techniques not only enhances the model's ability to detect code smells accurately but also mitigates the bias towards the majority class, resulting in a more balanced performance across different classes of code smells. This research has practical implications for software developers and researchers. It highlights the significance of considering data-balancing methods when applying ML models for code smells detection. By employing these methods, developers can enhance their ability to identify and address code quality issues, thereby improving software maintainability.

Chapter 8 Conclusion

8.1 Contributions

Identifying software bugs and code smells will help software developers distinguish code constructs that contain defects and assist them in the testing phase of the software development life cycle, resulting in improved software quality. Our dissertation contributes to software engineering, especially software bugs and code smell prediction. The main contribution of our dissertation is the development of different models based on several ML techniques combined with many data-balancing methods using software metrics to improve the prediction of software bugs and code smells. The criteria for selecting the ML techniques and data-balancing methods in this research work are based on the recommendations in the literature review. By making these contributions, our dissertation advances the understanding and application of data-balancing methods in the ML-based prediction of software bugs and code smells using software metrics. It provides valuable insights and practical guidance, aiding in developing more accurate and reliable prediction models and ultimately contributing to improving software quality and reliability. In summary, the main contributions of our research work are summarized as follows:

- Our dissertation makes a significant contribution by thoroughly examining the impact of the class imbalance problem on predicting software bugs and code smells. Where it provides insights into how class imbalance affects the performance of ML-based models and highlights the need for effective solutions to address this challenge.
- In this dissertation, we contribute by conducting a comprehensive evaluation of various data-balancing methods commonly employed to address the class imbalance problem in software bugs and code smells prediction.
- Our dissertation contributes to improving the accuracy and reliability of predictive models for software bugs and code smells by developing a novel prediction methodology based on ML techniques combined with data-balancing methods.
- In this dissertation, we validate our proposed methodology through experiments conducted on real-world software datasets, to show that the performance of ML algorithms in predicting software bugs and code smells can be significantly improved when balancing the data set by applying data-balancing methods. Additionally, this validation provides evidence of the effectiveness of the proposed methodology in practical settings and increases their applicability in real software development scenarios.

8.1.1 Theses - New Scientific Results

The dissertation presents results demonstrating the significant impact of class imbalance on the performance of predictive models. It highlights the challenges posed by class imbalance and provides empirical evidence of the effectiveness of data-balancing methods in enhancing the performance of predictive models for software bugs and code smells. The effectiveness of data-balancing methods in enhancing predictive models' performance is demonstrated through empirical evaluation based on Real-world software datasets using several standard performance measures. Overall, the dissertation presents new scientific results that contribute to data-balancing in ML-based prediction of software bugs and code smells using software metrics. The novel findings and evaluation results provide valuable insights and advance the understanding and application of data-balancing methods in improving the accuracy and

reliability of predictive models for software quality assurance. The main new scientific results of the research presented in this work are summarized in the following theses:

Thesis I: ***Investigating standard machine learning (ML) techniques previously used to predict software bugs and the impact of data-balancing methods (Undersampling methods) on the accuracy of ML models in software bug prediction (SBP).***

I proposed two approaches for SBP: in the first approach, I presented a comprehensive study investigating standard ML techniques previously used to predict software bugs. In addition, a method to examine the performance of classical supervised ML algorithms (DT, NB, RF, and LR) in SBP was proposed. The experiments were conducted based on four public benchmark datasets obtained from the NASA defect dataset. To investigate the impact of Undersampling methods in improving the accuracy of RNN models in SBP, a new approach was developed by combining two RNN models, namely LSTM and GRU, with an Undersampling method (Near Miss). The experiments were conducted on benchmark datasets which comprise five public datasets based on both class and file-level metrics. The results of both approaches were evaluated on many performance measures such as accuracy, precision, recall, f-measure, MCC, AUC, AUCPR, and MSE. Regarding the evaluation process and the results of the first approach: I established that the classic supervised ML algorithms can be used effectively for SBP. Regarding the experimental results of the second approach: the average Recall of the LSTM and GRU models on the original datasets (class level metrics and file level metrics) were 20 and 20%, and the average Recall of the models on the balanced datasets (class level metrics and file level metrics) using Near Miss were 92 and 81%. The results showed that the LSTM and GRU models on the balanced datasets improved the average Recall by 72 and 61%, respectively, compared to the original datasets. I established that there are positive effects of combining RNN with Undersampling methods on the performance of bug prediction regarding datasets with imbalanced class distributions and the proposed approaches are promising, competitive and suitable methodologies for SBP [P1 and P2].

Thesis II: ***Investigating the impact of data-balancing methods (Oversampling and hybrid sampling methods) on the accuracy of machine learning (ML) models in software defect prediction (SDP).***

I proposed two approaches to investigate the impact of Oversampling and hybrid sampling methods in improving the accuracy of advanced ML algorithms in SDP. The first approach was developed based on combining a Bi-LSTM network and Oversampling methods (Random Oversampling and SMOTE). The second approach was developed based on CNN and GRU combined with a hybrid sampling method (SMOTE Tomek). The experiments for both approaches have been conducted on benchmark datasets obtained from the PROMISE repository. The experimental results have been compared and evaluated in accuracy, precision, recall, f-measure, MCC, AUC, AUCPR, and MSE. Regarding the evaluation process and the results of the first approach: The average Recall of the Bi-LSTM model was 48% on the original datasets, 97% on balanced datasets (using Random Oversampling), and 94% on balanced datasets (using SMOTE). The results showed that the Bi-LSTM model on the balanced datasets improves the average Recall by 49 (using Random Oversampling) and 46% (using SMOTE), compared to the original datasets. Regarding the experimental results of the second approach: The average Recall of the CNN and GRU models were 48 and 49% on the original datasets and 94 and 91% on balanced datasets (using SMOTE Tomek), The results showed that the CNN and GRU models on the balanced datasets improve the average Recall

by 46 and 42%, respectively, compared to the original datasets. I established that combining advanced ML algorithms with Oversampling and hybrid sampling methods has positive effects on the performance of defect prediction regarding datasets with imbalanced class distributions. The proposed approaches are suitable methodologies for SDP [P3 and P4].

Thesis III: ***Investigating the impact of data-balancing methods (Oversampling and Undersampling methods) on the accuracy of machine learning (ML) models in code smells detection.***

I proposed three approaches to investigate the impact of Oversampling and Undersampling methods in improving the accuracy of classical and advanced ML algorithms in code smell detection. The first approach was developed based on five classic ML algorithms, namely DT, K-NN, SVM, XGB, and MLP combined with the Oversampling method (Random Oversampling). The second approach was developed based on a CNN combined with the Oversampling method (SMOTE). The third approach was developed based on two RNN models (Bi-LSTM and GRU) combined with Oversampling and Undersampling methods (Random Oversampling and Tomek links). The experiments for all approaches were conducted on four code smells datasets (God class, Data Class, Feature-envy, and Long-method) that were extracted from 74 open-source systems. The experimental results have been compared and evaluated in terms of accuracy, precision, recall, f-measure, MCC, AUC, AUCPR, and MSE. Regarding the evaluation process and the results of the first approach: The average Recall of the DT, K-NN, SVM, XGB and MLP models on the original datasets (God class, Data class, Long method and Feature envy) were 88, 95, 93 and 83%, respectively, and the average Recall of the models on the balanced datasets (using Random Oversampling) were 98, 99, 99 and 98%, respectively. The results showed that the DT, K-NN, SVM, XGB and MLP models on the balanced datasets improved the average Recall by 10, 4, 6 and 15%, respectively, compared to the original datasets. Regarding the evaluation process and the results of the second approach: the average Recall of the CNN model on the original datasets (God class, Data class, Feature envy and Long method) was 95%, and the average Recall of the model on the balanced datasets (using SMOTE) was 98%.The results showed that the CNN model on the balanced datasets improves the average Recall by 3%, compared to the original datasets. Regarding the experimental results of the third approach: the average Recall of the Bi-LSTM and GRU models were 91 and 92% on the original datasets (God class, Data class, Feature envy and Long method), the average Recall of the models were 99 and 99% on the balanced datasets using Random Oversampling, and the average Recall of the models were 96 and 96%, respectively, on the balanced datasets using Tomek links. The results showed that the Bi-LSTM and GRU models on the balanced datasets using Random Oversampling improved the average Recall by 8 and 7% and improved the average Recall by 5 and 4% on the balanced datasets using Tomek links, respectively, compared to the original datasets. I established that combining classic and advanced ML algorithms with Oversampling and Undersampling methods can improve the performance of code smell detection regarding datasets with imbalanced class distributions and the proposed approaches are suitable methodologies for code smell detection [P5, P6 and P7].

8.2 Future Research Direction

In terms of future research directions, our future research directions are summarized as follows:

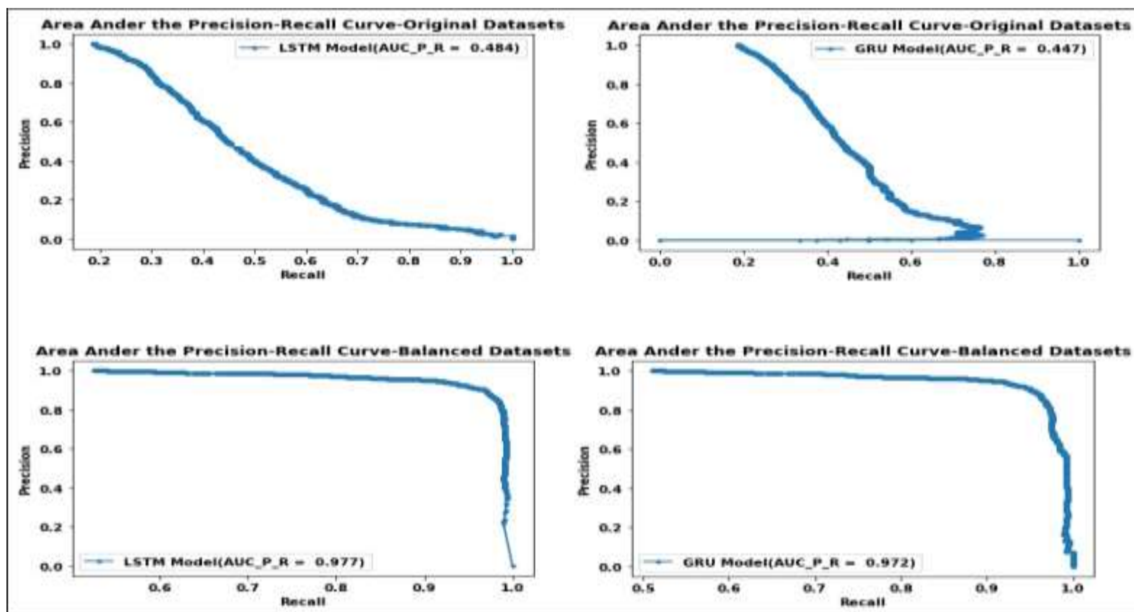
- Investigating advanced data-balancing methods: while this dissertation explores several commonly used data-balancing methods, future research can delve into more advanced techniques for addressing class imbalance in software bug and code smell prediction. This may include exploring ensemble-based methods, cost-sensitive learning approaches, or adaptive data-balancing techniques specifically tailored to the characteristics of software metrics.
- Hybrid approaches: future research can explore the potential of combining multiple data-balancing methods to achieve better performance in software bug and code smell prediction. Hybrid approaches may involve integrating Undersampling and Oversampling techniques, exploring the combination of synthetic and real data, or incorporating class weighting methods in conjunction with other data-balancing techniques.
- Handling multiclass imbalance: this dissertation primarily focuses on binary class imbalance, where the majority class dominates over the minority class. However, future research can explore the challenges and solutions for addressing multiclass imbalance in the context of software bugs and code smell prediction. This may involve developing new data-balancing methods or adapting existing techniques to handle multiple imbalanced classes effectively.
- Feature selection and dimensionality reduction: software metrics often encompass many features, which may lead to high-dimensional datasets. Future research can explore the impact of feature selection and dimensionality reduction techniques on data-balancing and predictive model performance. Investigating the effectiveness of different feature selection algorithms or dimensionality reduction methods in the presence of class imbalance can provide valuable insights.

As a future target, we also would like to address the limitations of this research and extend our developed models to be applied in another field in software engineering. By exploring these future research directions, researchers can further advance the field of data-balancing in ML-based prediction of software bugs and code smells using software metrics. These investigations will contribute to developing more sophisticated and effective approaches for addressing class imbalance, enhancing prediction accuracy, and improving software quality assurance practices.

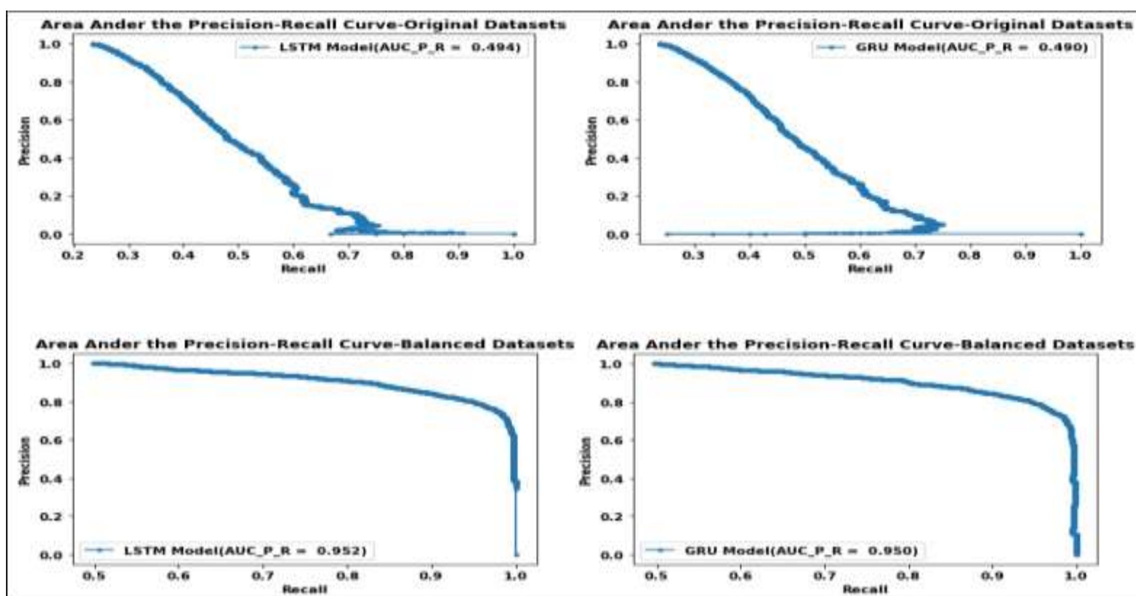
Appendices

Appendix 1: LSTM and GRU with Undersampling Methods in SBP

Figures 1 and 2 show the AUCPR scores obtained by the proposed models (LSTM and GRU) on the original and balanced datasets, respectively. The vertical axis presents the precision of the model, and the horizontal axis illustrates the recall.
Regarding the original datasets, the best AUCPR obtained by the both models (LSTM and GRU) which is 49% on the file level metrics dataset. While, the worst AUCPR obtained by GRU model which is 44% on the class level metrics dataset.
Regarding the balanced datasets, the best AUCPR obtained by the both models (LSTM and GRU) which is 97% on the class level metrics dataset. While, the worst AUCPR obtained by the both models (LSTM and GRU) which is 95% on the on the file level metrics dataset.



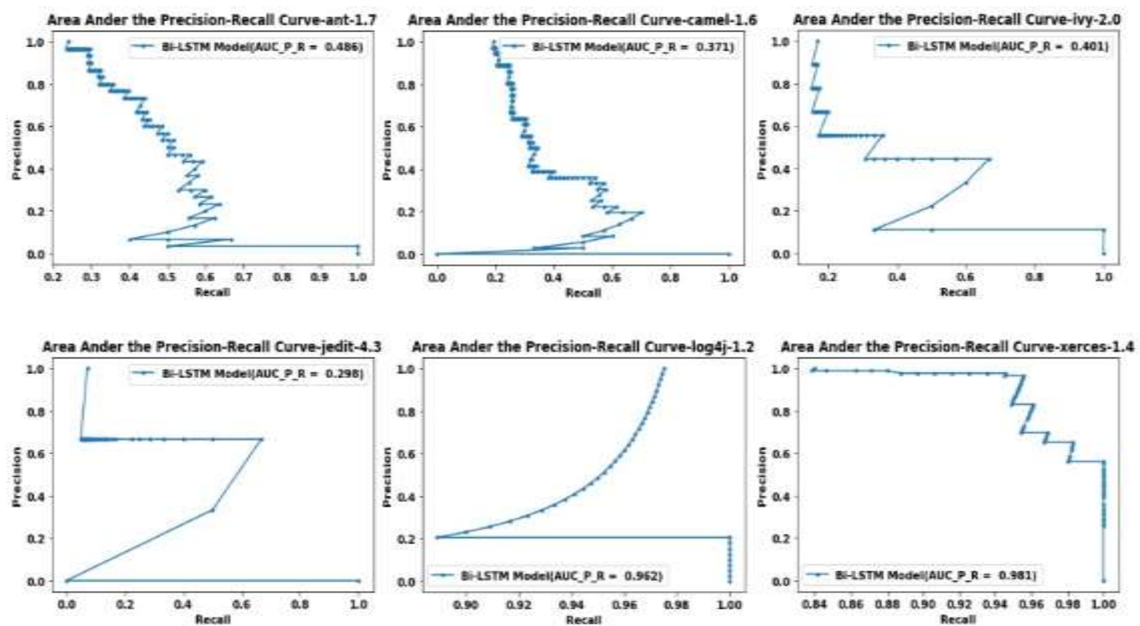
Appendix 1: 0.1 Figure 1. Illustrates the AUCPR of the models across all datasets - class-level metrics



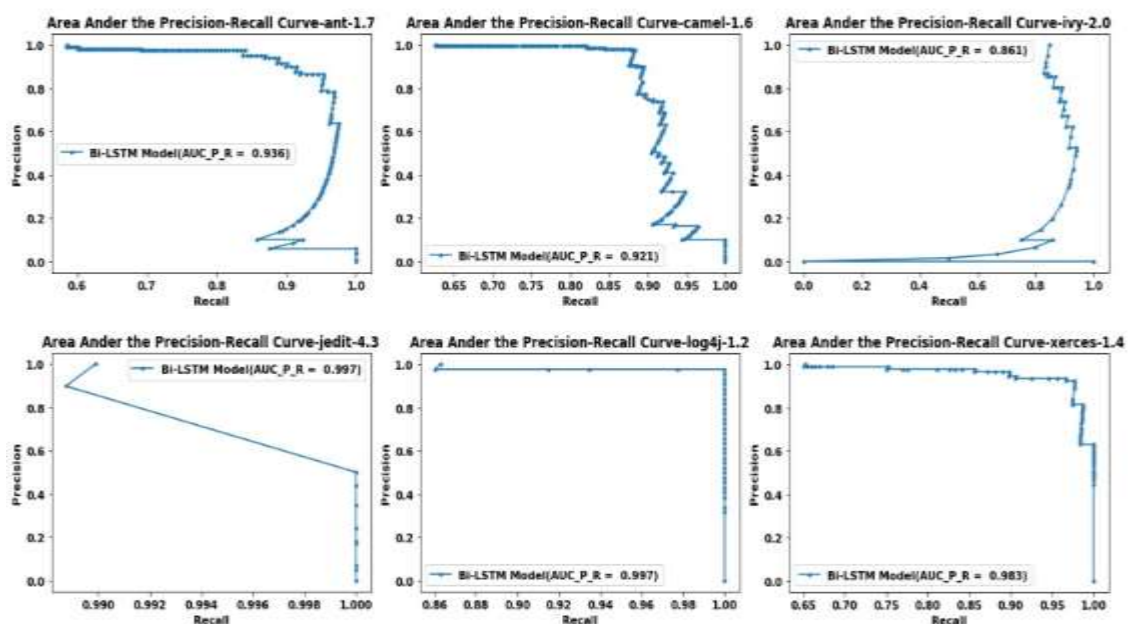
Appendix 1: 0.2 Figure 2. Illustrates the AUCPR of the models across all datasets - file-level metrics

Appendix 2: Bi-LSTM with Oversampling Methods in SDP

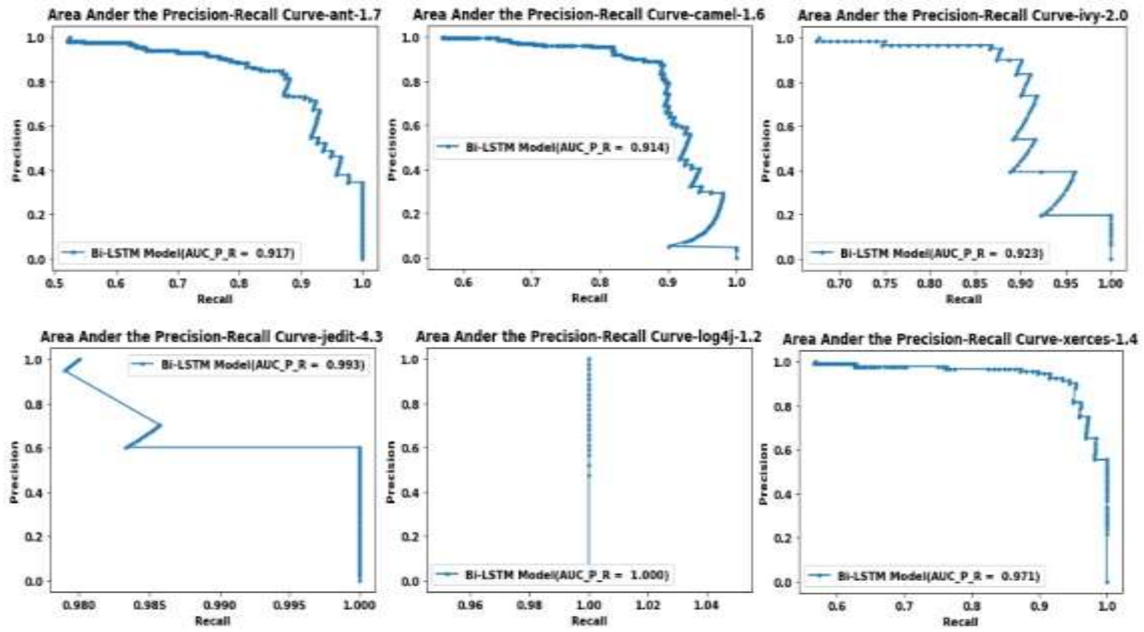
Figures 1, 2 and 3 show the AUCPR of the Bi-LSTM model on the original and balanced datasets.The vertical axis presents the precision of the model, and the horizontal axis illustrates the recall.
According to the Figures, the best AUCPR obtained by the proposed model in the original data sets is 98% on the xerces data set. The worst AUCPR is 29% on the jedit data set. The best AUCPR obtained by the proposed model in the balanced data sets (using Random Oversampling) is 99% on the jedit and log4j data sets, while the worst AUCPR is 86% on the ivy data set. The best AUCPR obtained by the proposed model in the balanced data sets (using SMOTE) is 100% on the log4j data set, while the worst AUCPR is 91% on the ant and camel data sets.



Appendix 2: 0.1 Figure 1. AUCPR for the original datasets



Appendix 2: 0.2 Figure 2. AUCPR for the balanced datasets - Random Oversampling

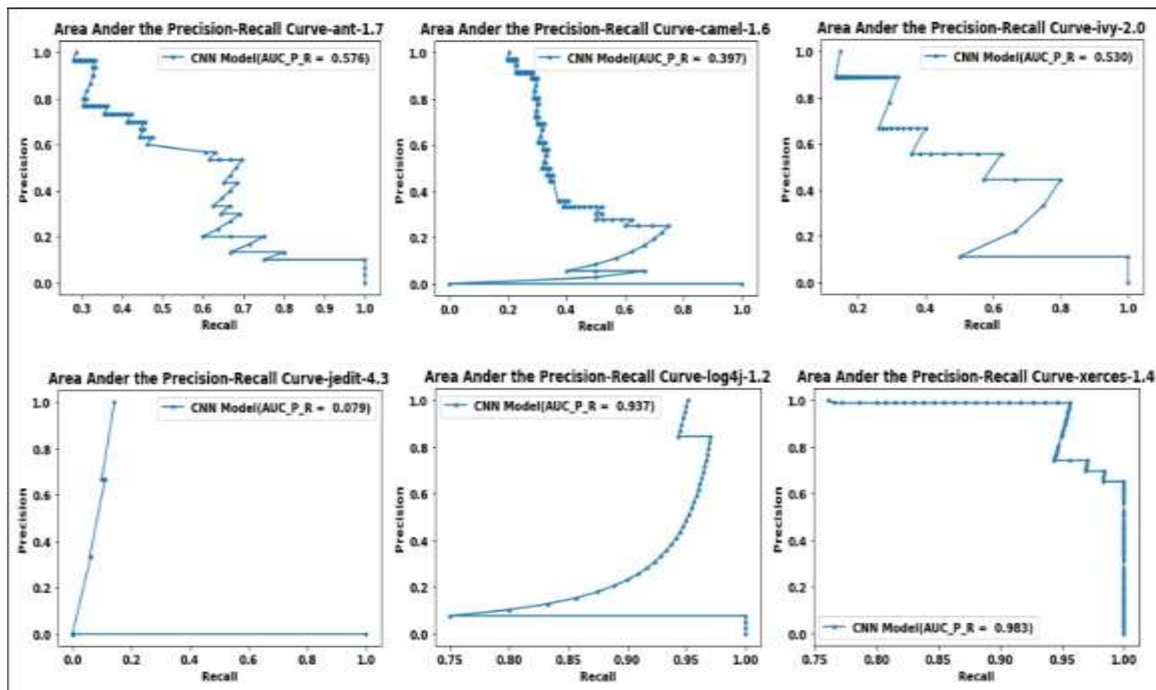


Appendix 2: 0.3 Figure 3. AUCPR for the balanced datasets – SMOTE

Appendix 3: CNN and GRU with Hybrid (Combined)-Sampling Methods in SDP

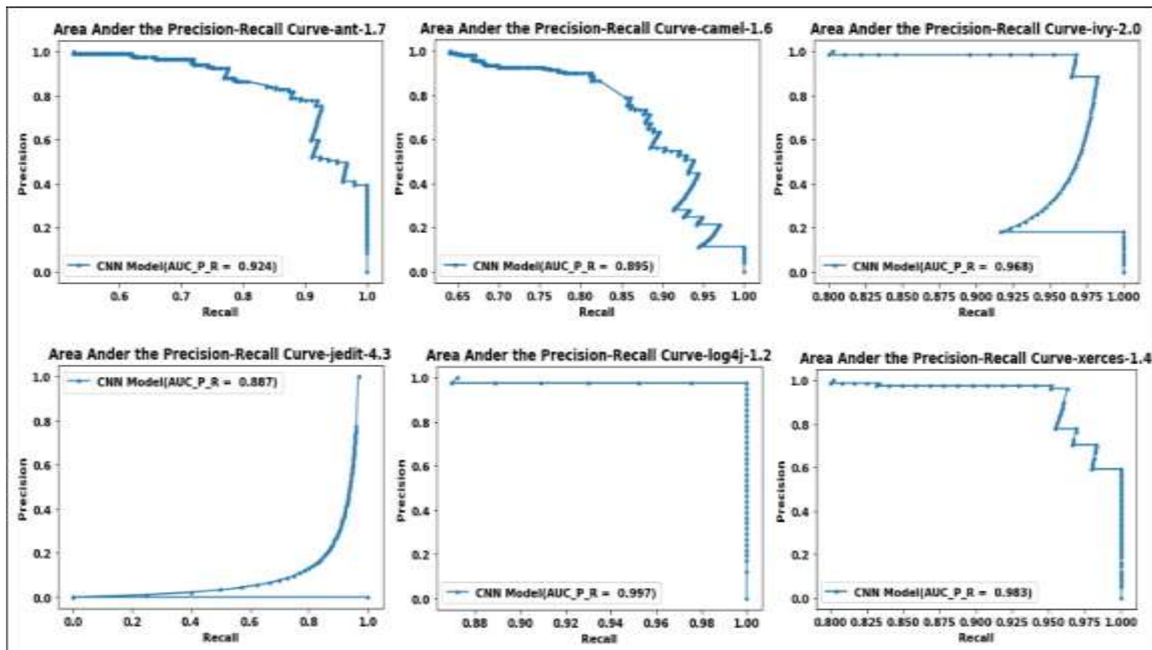
Figures 1 to 4 show the AUCPR of the proposed models (CNN and GRU) on the original and balanced datasets. The vertical axis presents the precision of the model, and the horizontal axis illustrates the recall.

Figure 1 shows the AUCPR values of the CNN model on the original data sets. The best AUCPR obtained is 98% on the *xerces* data set, while the worst AUCPR is 7% on the *jedit* data set.



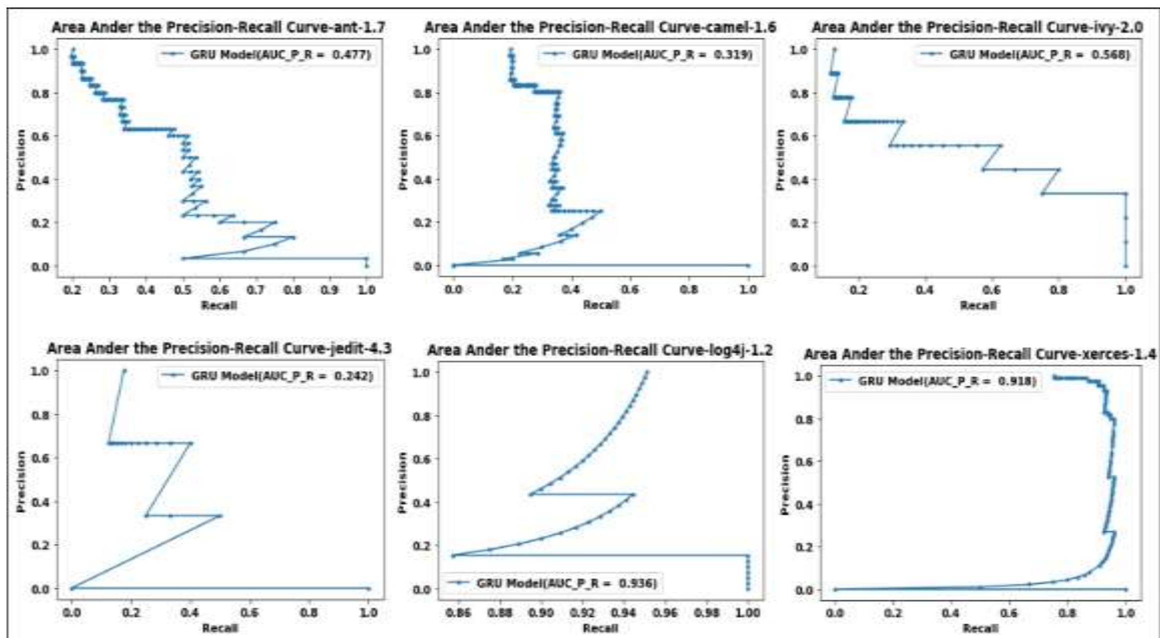
Appendix 3: 0.1 Figure 1. AUCPR for the original data sets - CNN model

Figure 2 shows the AUCPR values of the CNN model on the balanced data sets. The best AUCPR obtained is 99% on the *log4j* and *xerces* data sets, while the worst AUCPR is 88% on the *jedit* data set.



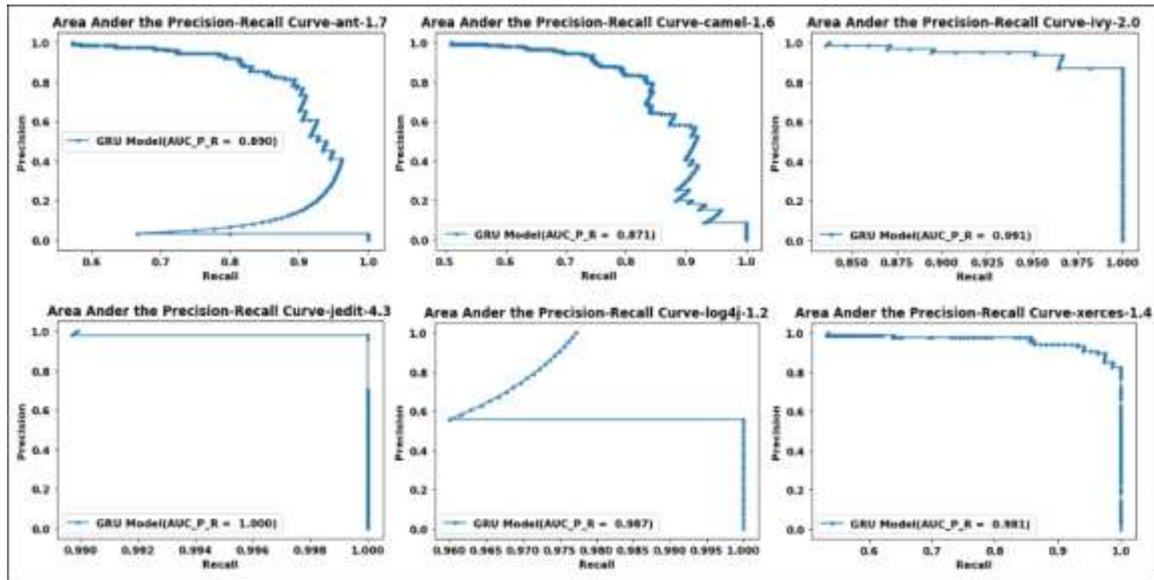
Appendix 3: 0.2 Figure 2. AUCPR for the balanced data sets - CNN model

Figure 3 shows the AUCPR values of the GRU model on the original data sets. The best AUCPR obtained is 93% on the log4j data set, while the worst AUCPR is 24% on the *jedit* data set.



Appendix 3: 0.3 Figure 3. AUCPR for the original data sets - GRU model

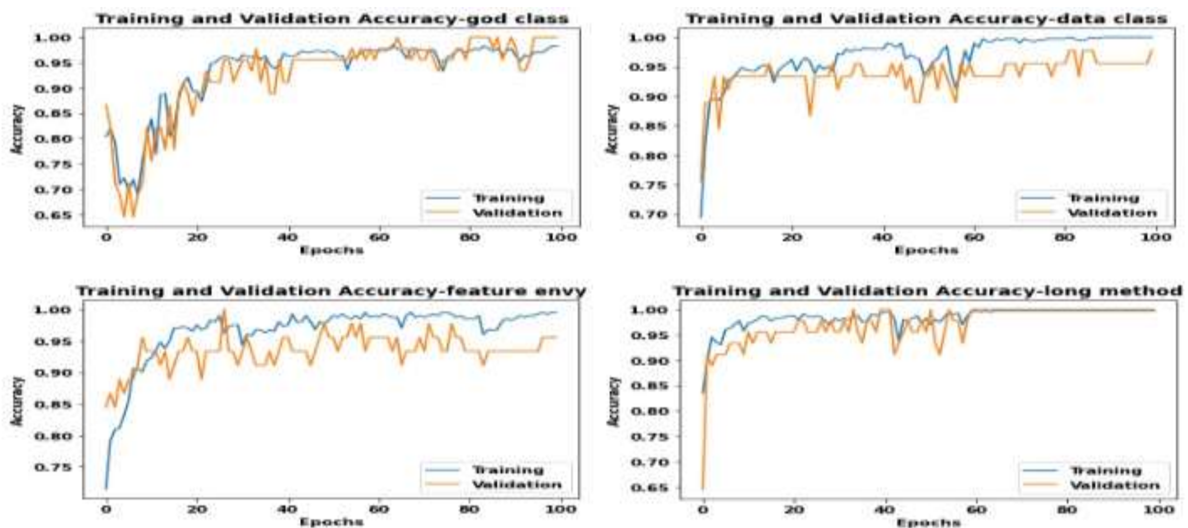
Figure 4 shows the AUCPR values of the GRU model on the balanced data sets. The best AUCPR obtained is 100% on the *jedit* data set, while the worst AUCPR is 84% on the *camel* data set.



Appendix 3: 0.4 Figure 4. AUCPR for the balanced data sets - GRU model

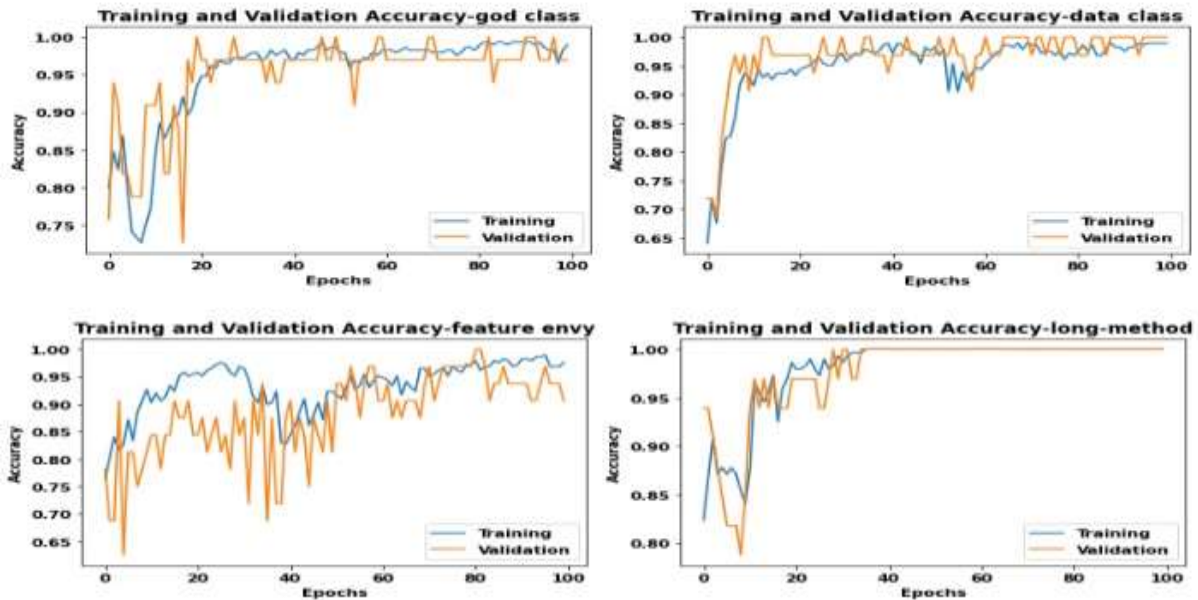
Appendix 4: Bi-LSTM and GRU with Under and Oversampling Methods

Figures 1 to 8 show the training and validation (accuracy and loss) of the proposed models (Bi-LSTM and GRU) on the balanced datasets.
Figures 1 to 4 show the training and validation accuracy of the models on the balanced datasets.The vertical axis presents the accuracy of the models, and the horizontal axis illustrates the number of epochs. Accuracy is the fraction of predictions that the models predicted right.
Figure 1 shows the accuracy values of the Bi-LSTM model with Random Oversampling technique. From the Figure, the model learned 96% accuracy for God Class, 99% accuracy for Data Class, 96% accuracy for Feature envy and 100% accuracy for Long method at the 100th epoch.



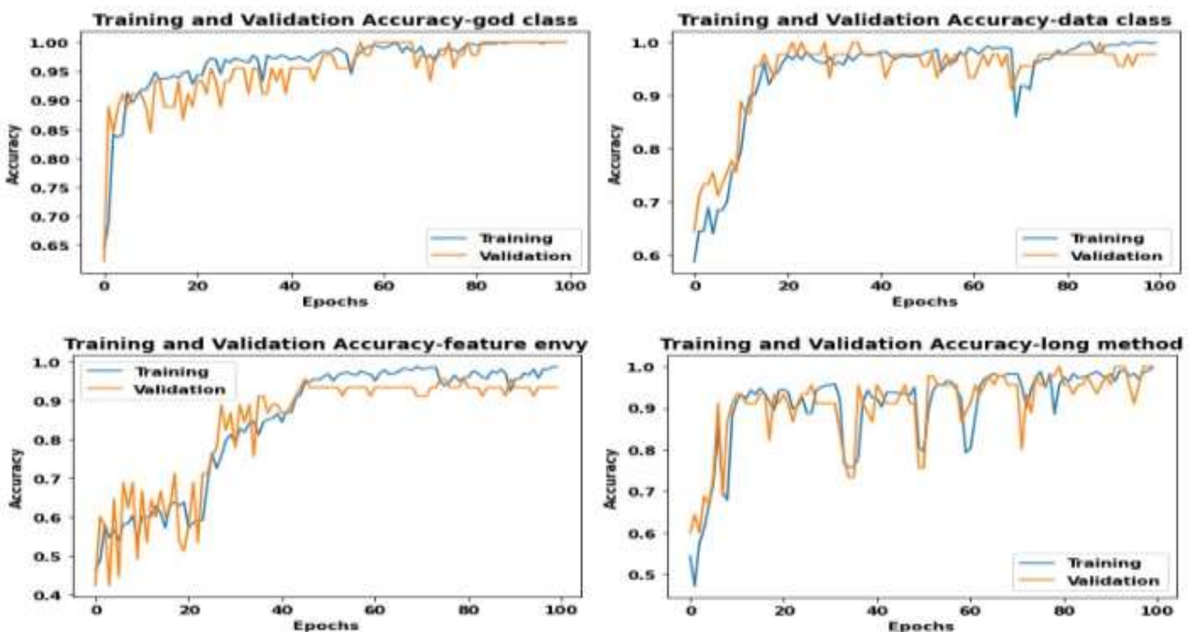
Appendix 4: 0.1 Figure 1. Training and Validation Accuracy on the balanced datasets using Bi-LSTM Model-Random Oversampling

Figure 2 shows the accuracy values of the Bi-LSTM model with Tomek links technique. From the Figure, the model learned 96% accuracy for God Class, 95% accuracy for Data Class, 98% accuracy for Feature envy and 99% accuracy for Long method at the 100th epoch.



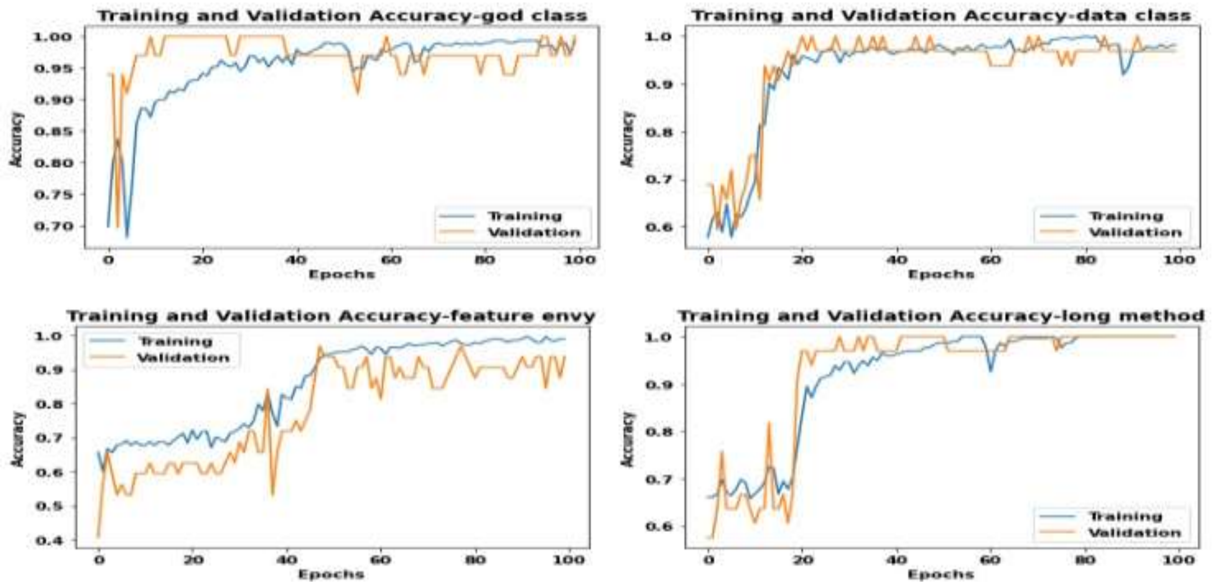
Appendix 4: 0.2 Figure 2. Training and Validation Accuracy on the balanced datasets using Bi-LSTM Model-Tomek links

Figure 3 shows the accuracy values of the GRU model with Random Oversampling technique. From the Figure, the model learned 96% accuracy for God Class, 98% accuracy for Data Class, 97% accuracy for Feature envy and 100% accuracy for Long method at the 100th epoch.



Appendix 4: 0.3 Figure 3. Training and Validation Accuracy on the balanced datasets using GRU Model-Random Oversampling

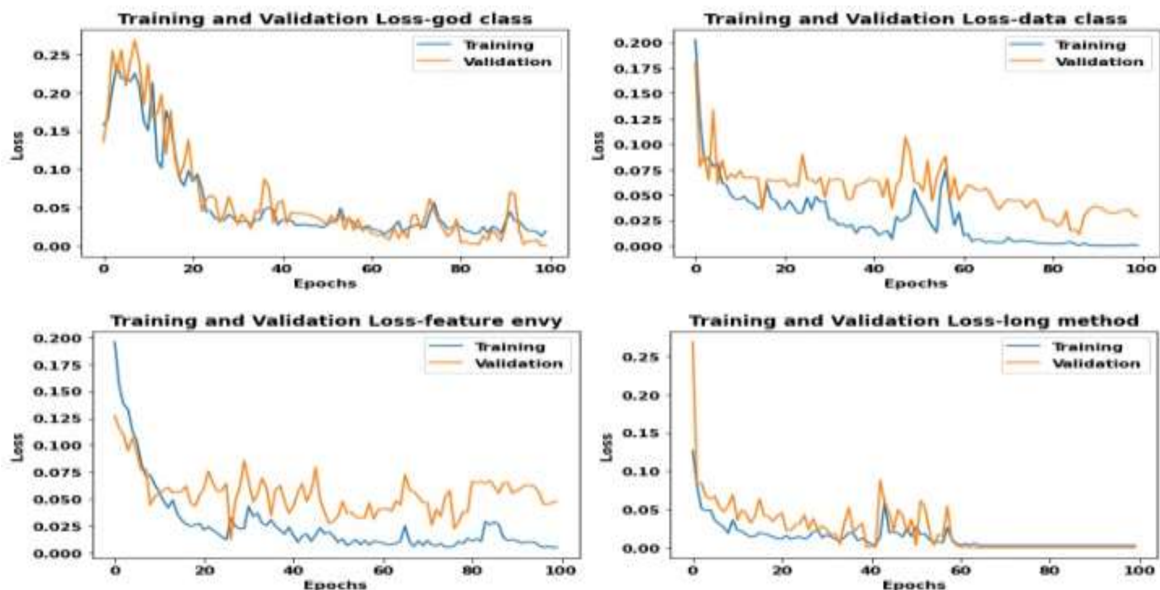
Figure 4 shows the accuracy values of the GRU model with Tomek links technique. From the Figure, the model learned 96% accuracy for God Class, 99% accuracy for Data Class, 99% accuracy for Feature envy and 98% accuracy for Long method at the 100th epoch.



Appendix 4: 0.4 Figure 4. Training and Validation Accuracy on the balanced datasets using GRU Model-Tomek links

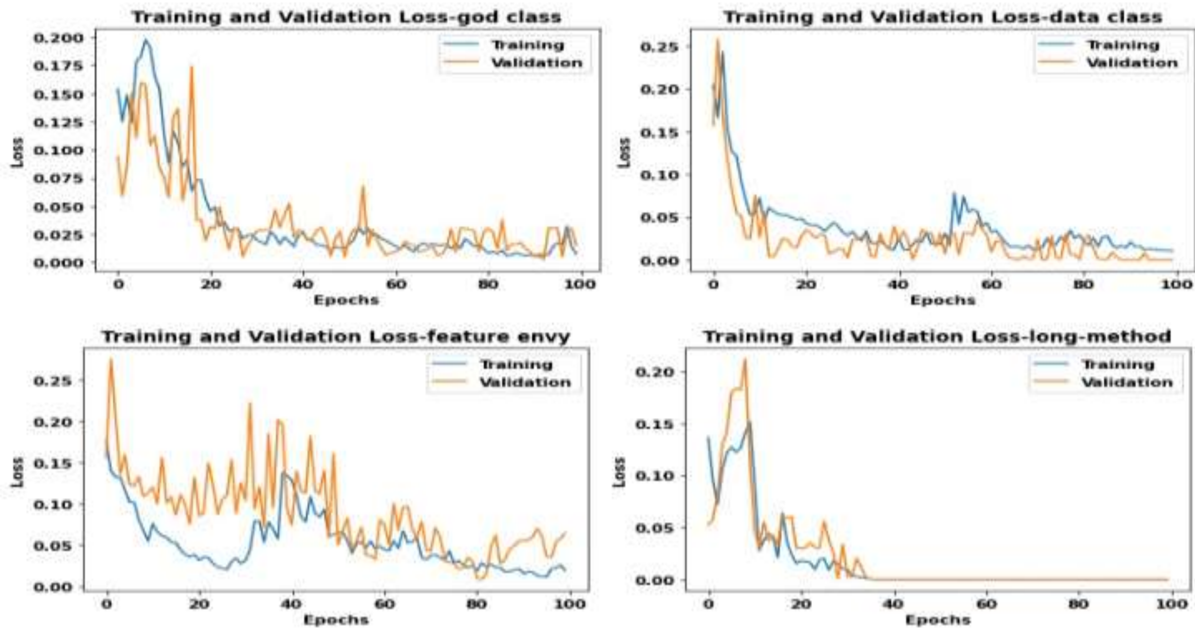
Figures 5 to 8 show the training and validation loss of the models on the balanced datasets. The vertical axis presents the loss of the models, and the horizontal axis illustrates the number of epochs. The loss indicates how bad a model's prediction was.

Figure 5 shows the loss values of the Bi-LSTM model with Random Oversampling technique. From the Figure, the model loss is 0.035 for God Class, 0.006 for Data Class, 0.037 for Feature envy and 0.005 for the long method at the 100th epoch.



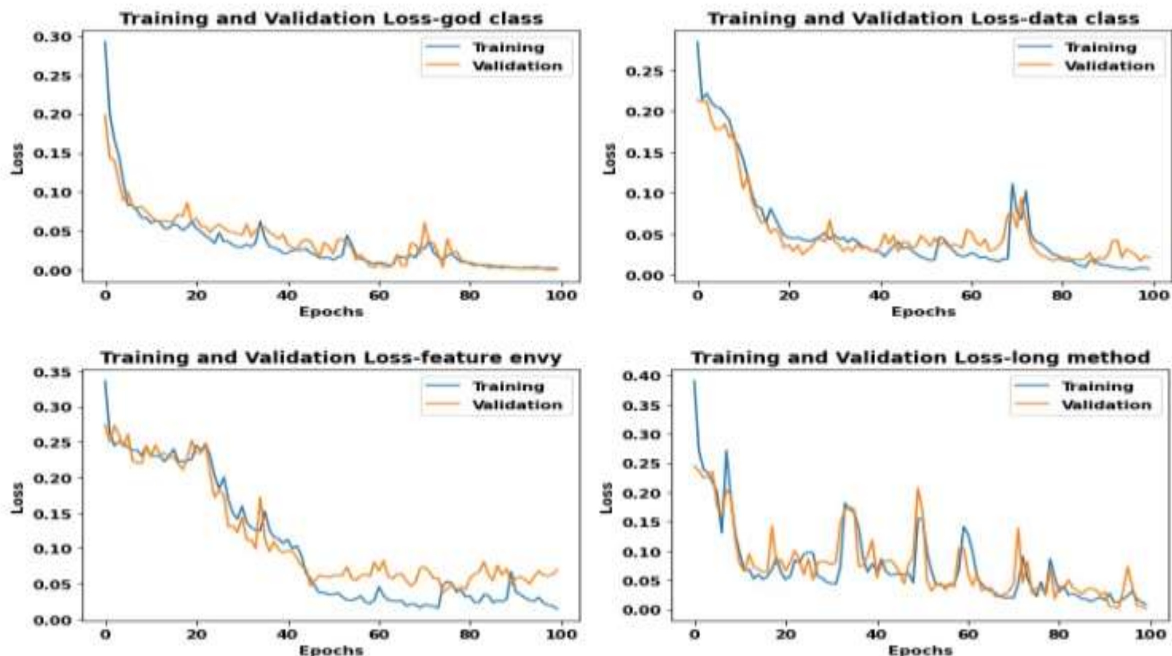
Appendix 4: 0.5 Figure 5. Training and Validation Loss on the balanced datasets using Bi-LSTM Model-Random Oversampling

Figure 6 shows the loss values of the Bi-LSTM model with Tomek links technique. From the Figure, the model loss is 0.037 for God Class, 0.044 for Data Class, 0.020 for Feature envy and 0.013 for the long method at the 100th epoch.



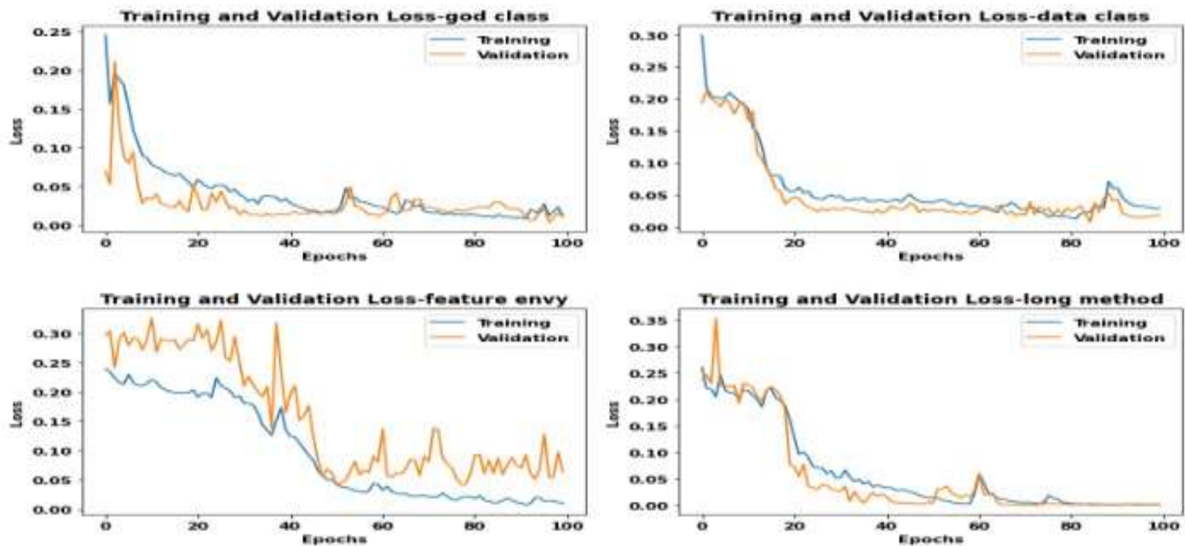
Appendix 4: 0.6 Figure 6. Training and Validation Loss on the balanced datasets using Bi-LSTM Model-Tomek links

Figure 7 shows the loss values of the GRU model with Random Oversampling technique. From the Figure, the model loss is 0.033 for God Class, 0.023 for Data Class, 0.032 for Feature envy and 0.002 for the long method at the 100th epoch.



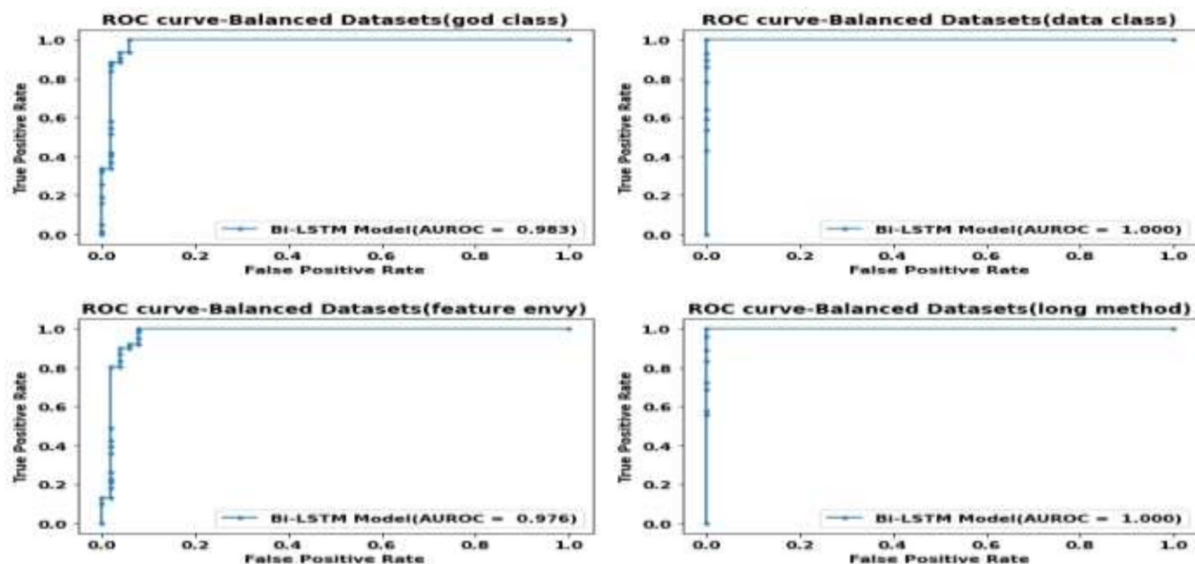
Appendix 4: 0.7 Figure 7. Training and Validation Loss on the balanced datasets using GRU Model-Random Oversampling

Figure 8 shows the loss values of the GRU model with Tomek links technique. From the Figure, the model loss is 0.038 for God Class, 0.018 for Data Class, 0.021 for Feature envy and 0.025 for the long method at the 100th epoch.
As shown in the Figures, the accuracy of training and validation increases, and the loss decreases with increasing epochs. Regarding the high accuracy and low loss obtained by the proposed models, we note that both models are well-trained and validated. Additionally, we note that the models are approximately perfectly fitting, there is no overfitting or underfitting.



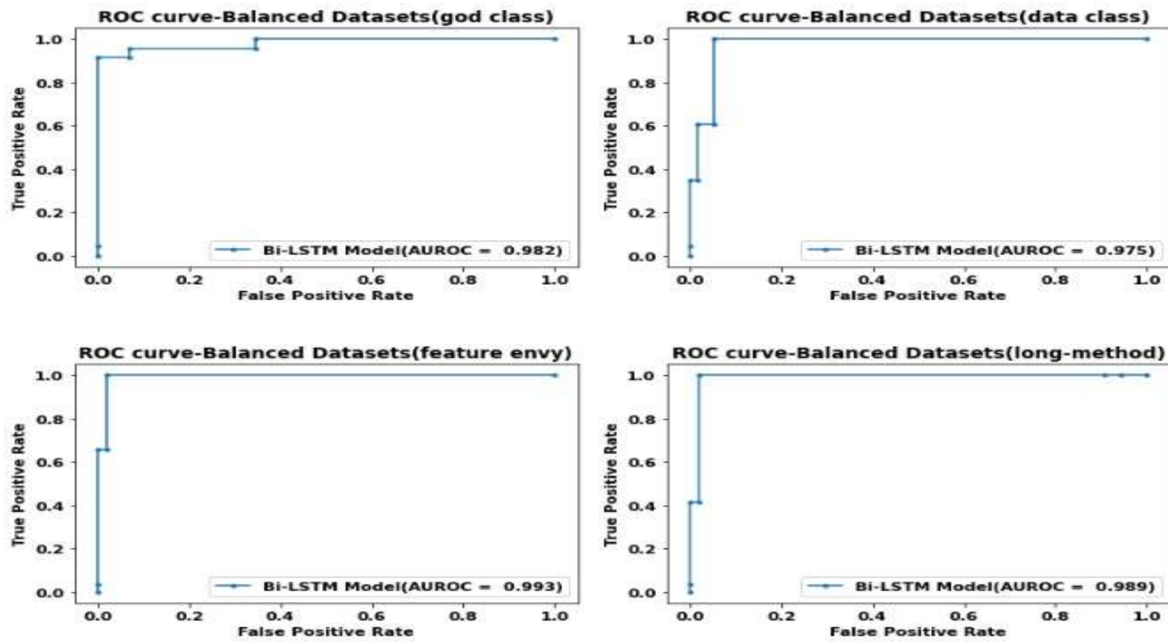
Appendix 4: 0.8 Figure 8. Training and Validation Loss on the balanced datasets using GRU Model - Tomek links

Furthermore, Figures 9 to 12 show the ROC curves for both models on the balanced datasets. The vertical axis presents the actual positive rate of the models, and the horizontal axis illustrates the false positive rate. The AUC is a sign of the performance of the model. The larger the AUC is, the better the model performance will be. Figure 9 shows the AUC values of the Bi-LSTM model with Random Oversampling technique. From the Figure, the AUC values are 98% on God Class,100% on Data Class, 97% on Feature envy and 100% on the Long method.



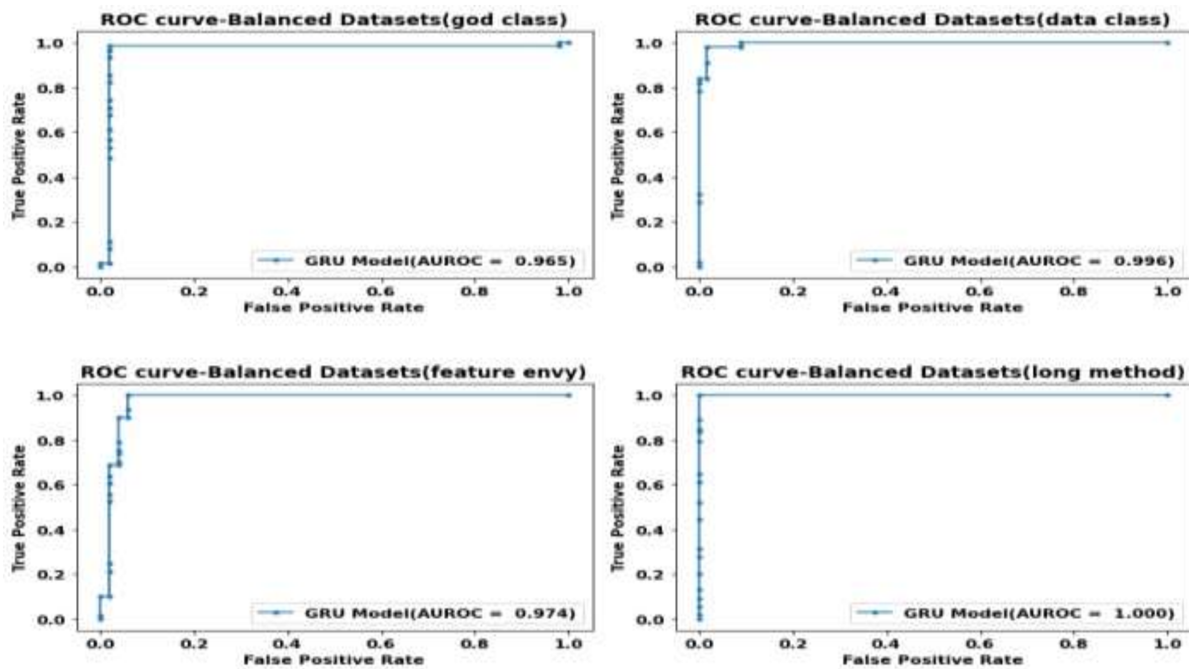
Appendix 4: 0.9 Figure 9. ROC curves for the balanced datasets - Bi-LSTM Model-Random Oversampling

Figure 10 shows the AUC values of the Bi-LSTM model with Tomek links technique. From the Figure, the AUC values are 0.98% on God Class,97% on Data Class, 99% on Feature envy and 98% on the Long method.



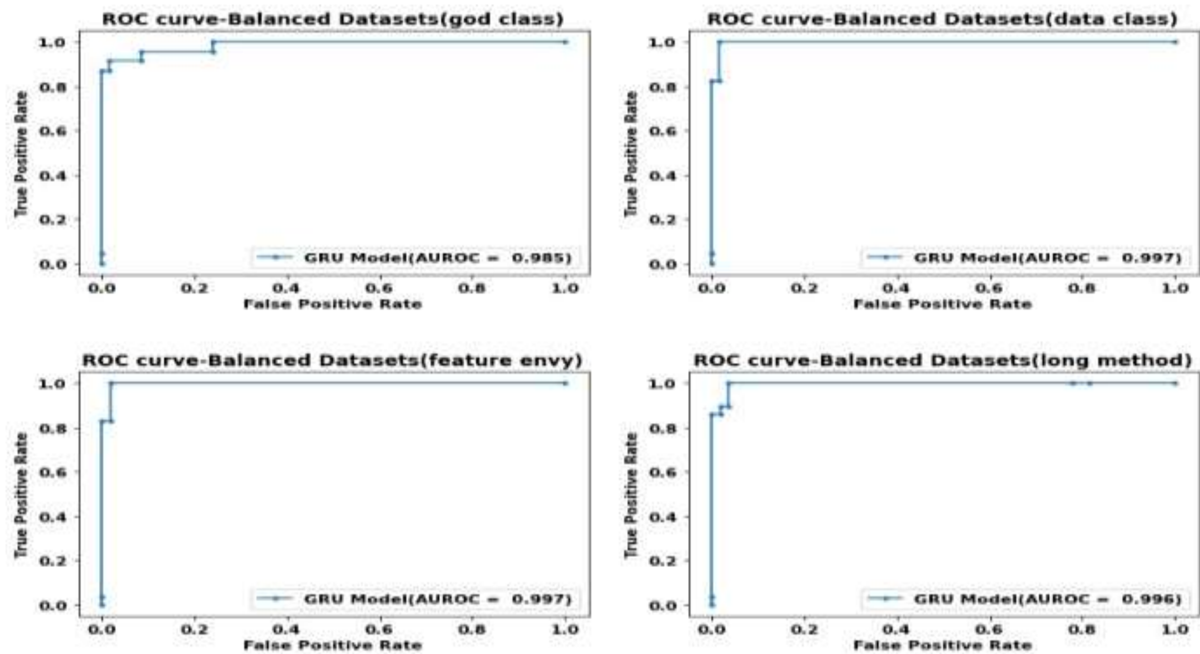
Appendix 4: 0.10 Figure 10. ROC curves for the balanced datasets - Bi-LSTM Model- Tomek links

Figure 11 shows the AUC values of the GRU model with Random Oversampling technique. From the Figure, the AUC values are 96% on God Class,99% on Data Class, 97% on Feature envy and 100% on the Long method.



Appendix 4: 0.11 Figure 11. ROC curves for the balanced datasets - GRU Model-Random Oversampling

Figure 12 shows the AUC values of the GRU model with Tomek links technique. From the Figure, the AUC values are 98% on God Class,99% on Data Class, 99% on Feature envy and 99% on the Long method.



Appendix 4: 0.12 Figure 12. ROC curves for the balanced datasets - GRU Model- Tomek links

Author's Publication

Publications Related to the Dissertation

(P1) **N. A. A. Khleel and K. Nehéz**, "Comprehensive Study on Machine Learning Techniques for Software Bug Prediction", International Journal of Advanced Computer Science and Applications,Vol.12,No.8,pp.726-735,2021. http://dx.doi.org/10.14569/IJACSA.2021.0120884. **Web of Science (WoS), Scopus (Q3), Impact Factor (1.16)**, Journal Article.

(P2) N.A.A.Khleel and K.Nehéz, "Improving the Accuracy of Recurrent Neural Networks Models in Predicting Software Bug Based on Undersampling Methods", Indonesian Journal of ElectricalEngineeringandComputerScience.Vol.32,No.1,pp.478-493,2023. http://doi.org/10.11591/ijeecs.v32.i1.pp478-493. **Scopus (Q3), Impact Factor (1.51)**, Journal Article.

(P3) N.A.A.Khleel and K.Nehéz, "Software Defect Prediction using a Bidirectional LSTM Network Combined with Oversampling Techniques", Cluster Computing (2023). https://doi.org/10.1007/s10586-023-04170-z. **Web of Science (WoS), Scopus (Q2), Impact Factor (4.4)**, Journal Article.

(P4) **N.A.A.Khleel and K.Nehéz**, "A novel approach for software defect prediction using CNN and GRU based on SMOTE Tomek method", Journal of Intelligent Information Systems (2023). https://doi.org/10.1007/s10844-023-00793-1.**Web of Science (WoS), Scopus (Q2), Impact Factor (3.4)**, Journal Article.

(P5) N.A.A.Khleel and K.Nehéz, "Detection of Code Smells Using Machine Learning Techniques Combined with Data-Balancing Methods", International Journal of Advances in Intelligent Informatics.Vol.9,No.3,pp.402-417,2023. https://doi.org/10.26555/ijain.v9i3.981. **Scopus (Q3), Impact Factor (1.88)**, Journal Article.

(P6) N.A.A.Khleel and K.Nehéz, "Deep convolutional neural network model for bad code smells detection based on oversampling method", Indonesian Journal of Electrical Engineering andComputerScience,Vol.26,No.3,pp.1725-1735,2022. http://doi.org/10.11591/ijeecs.v26.i3.pp1725-1735. **Scopus (Q3), Impact Factor (1.51)**, Journal Article.

(P7) N.A.A.Khleel and K.Nehéz, "Improving Accuracy of Code Smells Detection using a Bi-LSTM and GRU Networks with Data Balancing Techniques", International Journal of Data Science and Analytics, (under review). **Scopus (Q2)**, **Impact Factor (2.52),** Journal Article.

(P8) **N.A.A.Khleel and K.Nehéz**, "A new approach to software defect prediction based on convolutional neural network and bidirectional long short-term memory", Production Systems andInformationEngineering,Vol.10,No.3,pp.1-15,2022. https://doi.org/10.32968/psaie.2022.3.1. Journal Article.

(P9) **N.A.A.Khleel and K.Nehéz**, Data Balancing Methods in ML-Based Software Bug Prediction, Doktoranduszok Fóruma , (2022) pp. 59-67. Conference paper.

(P10) **N.A.A.Khleel and K.Nehéz**, Overview of modern software bug prediction approaches, Doktoranduszok Fóruma , (2021) pp. 55-61. Conference paper.

Other Publications Journal Articles and Conference Proceeding

(P11) **M.A.A.Mohammed, N.A.A.Khleel, N.P.Szabó et al**, "Modeling of groundwater quality index by using artificial intelligence algorithms in northern Khartoum State, Sudan", Model. Earth Syst. Environ, 9, 2501–2516 (2023). https://doi.org/10.1007/s40808-022-01638-6. **Web of Science (WoS), Scopus (Q1), Impact Factor (3.90)**, Journal Article.

(P12) **N.A.A.Khleel and K.Nehéz**, "Merging problems in modern version control systems ", MultidisciplinarySciences,Vol.10,No.3,pp.365-376,2020. https://doi.org/10.35925/j.multi.2020.3.44. Journal Article.

(P13) **N.A.A.Khleel and K.Nehéz**, "Comparison of version control system tools", MultidisciplinarySciences,Vol.10,No.3,pp.61-69,2020. https://doi.org/10.35925/j.multi.2020.3.7. Journal Article.

(P14) **N.A.A.Khleel and K.Nehéz**, "Tools, processes and factors influencing code review ", MultidisciplinarySciences,Vol.10,No.3,pp.277-284,2020. https://doi.org/10.35925/j.multi.2020.3.33. Journal Article.

(P15**) N.A.A.Khleel and K.Nehéz**, Mining Software Repository: an Overview, Doktoranduszok Fóruma , (2019) pp. 108-114. Conference paper.

References

[1] G. Sharma, S. Sharma, and S. Gujral, "A Novel Way of Assessing Software Bug Severity Using Dictionary of Critical Terms," in *Procedia Computer Science*, Elsevier B.V., 2015, pp. 632–639. doi: 10.1016/j.procs.2015.10.059.

[2] H. Bani-Salameh, M. Sallam, and B. Al shboul, "A deep-learning-based bug priority prediction using RNN-LSTM neural networks," *E-Informatica Software Engineering Journal*, vol. 15, no. 1, pp. 29–45, 2021, doi: 10.37190/E-INF210102.

[3] A. Majd, M. Vahidi-Asl, A. Khalilian, P. Poorsarvi-Tehrani, and H. Haghighi, "SLDeep: Statement-level software defect prediction using deep-learning model on static code features," *Expert Syst Appl*, vol. 147, Jun. 2020, doi: 10.1016/j.eswa.2019.113156.

[4] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empir Softw Eng*, vol. 21, no. 3, pp. 1143–1191, Jun. 2016, doi: 10.1007/s10664-015-9378-4.

[5] A. Al-Shaaby, H. Aljamaan, and M. Alshayeb, "Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review," *Arabian Journal for Science and Engineering*, vol. 45, no. 4. Springer, pp. 2341–2369, Apr. 01, 2020. doi: 10.1007/s13369-019-04311-w.

[6] P. Kokol, M. K. Semantika, S. Zagoranski, and M. Kokol, "Code smells: A Synthetic Narrative Review Code smells: A Synthetic Narrative Review Code smells: A Synthetic Narrative Review," 2020. [Online]. Available: https://digitalcommons.unl.edu/libphilprac

[7] N. A. A. Khleel and K. Nehéz, "A novel approach for software defect prediction using CNN and GRU based on SMOTE Tomek method," *J Intell Inf Syst*, Jun. 2023, doi: 10.1007/s10844-023-00793-1.

[8] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, Oct. 2019, doi: 10.1145/3360588.

[9] L. Jonsson, M. Borg, D. Broman, K. Sandahl, S. Eldh, and P. Runeson, "Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts," *Empir Softw Eng*, vol. 21, no. 4, pp. 1533–1578, Aug. 2016, doi: 10.1007/s10664-015-9401-9.

[10] S. Aleem, L. Fernando Capretz, and F. Ahmed, "COMPARATIVE PERFORMANCE ANALYSIS OF MACHINE LEARNING TECHNIQUES FOR SOFTWARE BUG DETECTION," pp. 71–79, 2015, doi: 10.5121/csit.2015.50108.

[11] H. Tong, B. Liu, and S. Wang, "Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning," *Inf Softw Technol*, vol. 96, pp. 94–111, Apr. 2018, doi: 10.1016/j.infsof.2017.11.008.

[12] N. Moha, Y. G. Guéhéneuc, L. Duchien, and A. F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010, doi: 10.1109/TSE.2009.50.

[13] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, "On the role of data balancing for machine learning-based code smell detection," in *MaLTeSQuE 2019 - Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, co-located with ESEC/FSE 2019*, Association for Computing Machinery, Inc, Aug. 2019, pp. 19–24. doi: 10.1145/3340482.3342744.

[14] N. A. A. Khleel and K. Nehéz, "Deep convolutional neural network model for bad code smells detection based on oversampling method," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 26, no. 3, pp. 1725–1735, Jun. 2022, doi: 10.11591/ijeecs.v26.i3.pp1725-1735.

[15] M. Gao, X. Hong, S. Chen, C. J. Harris, and E. Khalaf, "PDFOS: PDF estimation based over-sampling for imbalanced two-class problems," *Neurocomputing*, vol. 138, pp. 248–259, Aug. 2014, doi: 10.1016/j.neucom.2014.02.006.

[16] U. Ali, S. Aftab, A. Iqbal, Z. Nawaz, M. S. Bashir, and M. A. Saeed, "Software defect prediction using variant based ensemble learning and feature selection techniques," *International Journal of Modern Education and Computer Science*, vol. 12, no. 5, pp. 29–40, 2020, doi: 10.5815/ijmecs.2020.05.03.

[17] N. A. A. Khleel and K. Nehéz, "Software defect prediction using a bidirectional LSTM network combined with oversampling techniques," *Cluster Computing* (2023). https://doi.org/10.1007/s10586-023-04170-z.

[18] N. A. A. Khleel and K. Nehéz, "Comprehensive Study on Machine Learning Techniques for Software Bug Prediction." [Online]. Available: www.ijacsa.thesai.org

[19] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Inf Softw Technol*, vol. 58, pp. 388–402, Feb. 2015, doi: 10.1016/j.infsof.2014.07.005.

[20] E. ÖZTÜRK, K. U. Birant, and D. Birant, "Yazılım Hata Tahmini için Sıralı Sınıflandırma Yaklaşımı," *Deu Muhendislik Fakultesi Fen ve Muhendislik*, vol. 21, no. 62, pp. 533–544, May 2019, doi: 10.21205/deufmd.2019216218.

[21] A. Hammouri, M. Hammad, M. Alnabhan, and F. Alsarayrah, "Software Bug Prediction using machine learning approach," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 2, pp. 78–83, 2018, doi: 10.14569/IJACSA.2018.090212.

[22] M. Efendioglu, A. Sen, and Y. Koroglu, "Bug prediction of systemC models using machine learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 3, pp. 419–429, Mar. 2019, doi: 10.1109/TCAD.2018.2878193.

[23] J. A. Fadhil, K. T. Wei, and K. S. Na, "Artificial Intelligence for Software Engineering: An Initial Review on Software Bug Detection and Prediction," *Journal of Computer Science*, vol. 16, no. 12, pp. 1709–1717, 2020, doi: 10.3844/jcssp.2020.1709.1717.

[24] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the NASA software defect datasets," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1208–1215, 2013, doi: 10.1109/TSE.2013.11.

[25] A. Professor, "Overview of Software Defect Prediction using Machine Learning Algorithms." [Online]. Available: http://www.ijpam.eu

[26] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108. Elsevier B.V., pp. 115–138, Apr. 01, 2019. doi: 10.1016/j.infsof.2018.12.009.

[27] M. Y. Mhawish and M. Gupta, "Predicting Code Smells and Analysis of Predictions: Using Machine Learning Techniques and Software Metrics," *J Comput Sci Technol*, vol. 35, no. 6, pp. 1428–1445, Nov. 2020, doi: 10.1007/s11390-020-0323-7.

[28] M. Hadj-Kacem and N. Bouassida, "A hybrid approach to detect code smells using deep learning," in *ENASE 2018 - Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering*, SciTePress, 2018, pp. 137–146. doi: 10.5220/0006709801370146.

[29] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, "Deep learning based code smell detection," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1811–1837, Sep. 2021, doi: 10.1109/TSE.2019.2936376.

[30] D. Oliveira, W. K. G. Assunção, L. Souza, W. Oizumi, A. Garcia, and B. Fonseca, "Applying Machine Learning to Customized Smell Detection: A Multi-Project Study," in *ACM International Conference Proceeding Series*, Association for Computing Machinery, Oct. 2020, pp. 233–242. doi: 10.1145/3422392.3422427.

[31] M. Gradišnik, T. Beranič, S. Karakatič, and G. Mauša, "Adapting God Class thresholds for software defect prediction: A case study." [Online]. Available: https://projects.eclipse.org/

[32] G. Saranya, H. Khanna Nehemiah, A. Kannan, and V. Nithya, "Model level code smell detection using EGAPSO based on similarity measures," *Alexandria Engineering Journal*, vol. 57, no. 3. Elsevier B.V., pp. 1631–1642, Sep. 01, 2018. doi: 10.1016/j.aej.2017.07.006.

[33] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Software Quality Journal*, vol. 25, no. 2, pp. 529–552, Jun. 2017, doi: 10.1007/s11219-016-9309-7.

[34] D. K. Kim, "Finding bad code smells with neural network models," *International Journal of Electrical and Computer Engineering*, vol. 7, no. 6, pp. 3613–3621, Dec. 2017, doi: 10.11591/ijece.v7i6.pp3613-3621.

[35] F. Caram Luiz, B. Rafael de Oliveira Rodrigues, and F. Silva Parreiras, "Machine learning techniques for code smells detection: an empirical experiment on a highly imbalanced setup," 2019, doi: 10.1145/3330204.

[36] F. Arcelli Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowl Based Syst*, vol. 128, pp. 43–58, Jul. 2017, doi: 10.1016/j.knosys.2017.04.014.

[37] F. L. Caram, B. R. D. O. Rodrigues, A. S. Campanelli, and F. S. Parreiras, "Machine Learning Techniques for Code Smells Detection: A Systematic Mapping Study," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 2, pp. 285–316, Feb. 2019, doi: 10.1142/S021819401950013X.

[38] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *IEEE International Conference on Program Comprehension*, IEEE Computer Society, May 2019, pp. 93–104. doi: 10.1109/ICPC.2019.00023.

[39] N. A. A. Khleel and K. Nehéz, "Detection of code smells using machine learning techniques combined with data-balancing methods," *International Journal of Advances in Intelligent Informatics*, vol. 9, no. 3, pp. 402-417, 2023, doi: https://doi.org/10.26555/ijain.v9i3.981.

[40] N. A. A. Khleel and K. Nehéz, "Improving the accuracy of recurrent neural networks models in predicting software bug based on undersampling methods," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 32, no. 1, p. 478, Oct. 2023, doi: 10.11591/ijeecs.v32.i1.pp478-493.

[41] S. Puranik, P. Deshpande, and K. Chandrasekaran, "A Novel Machine Learning Approach for Bug Prediction," in *Procedia Computer Science*, Elsevier B.V., 2016, pp. 924–930. doi: 10.1016/j.procs.2016.07.271.

[42] V. Gupta, N. Ganeshan, and T. K. Singhal, "Developing Software Bug Prediction Models Using Various Software Metrics as the Bug Indicators," 2015. [Online]. Available: www.ijacsa.thesai.org

[43] S. Karim, H. Leslie Hendric Spits Warnars, F. Lumban Gaol, E. Abdurachman, and B. Soewito, "Software Metrics for Fault Prediction Using Machine Learning Approaches A Literature Review with PROMISE Repository Dataset."

[44] S. N. A. Saharudin, K. T. Wei, and K. S. Na, "Machine Learning Techniques for Software Bug Prediction: A Systematic Review," *Journal of Computer Science*, vol. 16, no. 11, pp. 1558–1569, 2020, doi: 10.3844/JCSSP.2020.1558.1569.

[45] D. I. G. Amalarethinam, P. H. Maitheen, and S. Hameed, "Analysis of Object Oriented Metrics on a Java Application," 2015.

[46] R. Suresh Kumar and B. Satyanarayana, "Adaptive Genetic Algorithm Based Artificial Neural Network for Software Defect Prediction," *Type: Double Blind Peer Reviewed International Research Journal Publisher: Global Journals Inc*, vol. 15, 2015.

[47] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "HYDRA: Massively compositional model for cross-project defect prediction," *IEEE Transactions on Software Engineering*, vol. 42, no. 10, pp. 977–998, Oct. 2016, doi: 10.1109/TSE.2016.2543218.

[48] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A Semantic LSTM Model for Software Defect Prediction," *IEEE Access*, vol. 7, pp. 83812–83824, 2019, doi: 10.1109/ACCESS.2019.2925313.

[49] R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth, and T. Gyimóthy, "An automatically created novel bug dataset and its validation in bug prediction," *Journal of Systems and Software*, vol. 169, Nov. 2020, doi: 10.1016/j.jss.2020.110691.

[50] F. A. Batarseh, A. Kumar, R. Mohod, and J. Bui, "Chapter 10: The Application of Artificial Intelligence in Software Engineering-A Review Challenging Conventional Wisdom."

[51] D. R. Prashanta and K. Patra, "LECTURE NOTES ON ARTIFICIAL INTELLIGENCE PREPARED BY."

[52] F. Meziane and S. Vadera, "Artificial Intelligence in Software Engineering," 2010, pp. 278–299. doi: 10.4018/978-1-60566-758-4.ch014.

[53] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A Comprehensive Study on Deep Learning Bug Characteristics," Jun. 2019, [Online]. Available: http://arxiv.org/abs/1906.01388

[54] F. Qin, X. Wan, and B. Yin, "An empirical study of factors affecting cross-project aging-related bug prediction with TLAP," *Software Quality Journal*, vol. 28, no. 1, pp. 107–134, Mar. 2020, doi: 10.1007/s11219-019-09460-7.

[55] X. Ye, F. Fang, J. Wu, R. Bunescu, and C. Liu, "Bug Report Classification Using LSTM Architecture for More Accurate Software Defect Locating," in *Proceedings - 17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018*, Institute of Electrical and Electronics Engineers Inc., Jan. 2019, pp. 1438–1445. doi: 10.1109/ICMLA.2018.00234.

[56] S. Sah, "Machine Learning: A Review of Learning Types," 2020, doi: 10.20944/preprints202007.0230.v1.

[57] B. Mahesh, "Machine Learning Algorithms-A Review Machine Learning Algorithms-A Review View project Self Flowing Generator View project Batta Mahesh Independent Researcher Machine Learning Algorithms-A Review," *International Journal of Science and Research*, 2018, doi: 10.21275/ART20203995.

[58] T. Oladipupo Ayodele, "X Types of Machine Learning Algorithms." [Online]. Available: www.intechopen.com

[59] R. Kumar and S. Singla, "Multiclass Software Bug Severity Classification using Decision Tree, Naive Bayes and Bagging," 2021.

[60] A. Kukkar, R. Mohana, A. Nayyar, J. Kim, B. G. Kang, and N. Chilamkurti, "A novel deep-learning-based bug severity classification technique using convolutional neural networks and random forest with boosting," *Sensors (Switzerland)*, vol. 19, no. 13, Jul. 2019, doi: 10.3390/s19132964.

[61] A. Baarah, A. Aloqaily, Z. Salah, M. Zamzeer, and M. Sallam, "Machine Learning Approaches for Predicting the Severity Level of Software Bug Reports in Closed Source Projects," 2019. [Online]. Available: www.ijacsa.thesai.org

[62] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, "How bugs are born: a model to identify how bugs are introduced in software components," *Empir Softw Eng*, vol. 25, no. 2, pp. 1294–1340, Mar. 2020, doi: 10.1007/s10664-019-09781-y.

[63] S. Jain and A. Saha, "Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection," *Sci Comput Program*, vol. 212, Dec. 2021, doi: 10.1016/j.scico.2021.102713.

[64] S. Sharma and S. Kumar, “Analysis of ensemble models for aging related bug prediction in software systems,” in *ICSOFT 2018 - Proceedings of the 13th International Conference on Software Technologies*, SciTePress, 2019, pp. 256–263. doi: 10.5220/0006847702560263.

[65] A. Abraham, Machine Intelligence Research Labs, M. and C. S. T. C. on S. C. Systems, Annual IEEE Computer Conference, International Conference on Intelligent Systems Design and Applications 12 2012.11.27-29 Kochi, and ISDA 12 2012.11.27-29 Kochi, *12th International Conference on Intelligent Systems Design and Applications (ISDA), 2012 27-29 Nov. 2012, Kochi, India*.

[66] F. Qin, Z. Zheng, Y. Qiao, and K. S. Trivedi, “Studying Aging-Related Bug Prediction Using Cross-Project Models,” *IEEE Trans Reliab*, Sep. 2018, doi: 10.1109/TR.2018.2864960.

[67] A. Abdou, F. Akmel, and E. Birihanu, “A Literature Review Study of Soware Defect Prediction using Machine Learning Techniques Related papers Early Predict ion of Soft ware Defect using Ensemble Learning: A Comparat ive St udy A Literature Review Study of Software Defect Prediction using Machine Learning Techniques,” 2017. [Online]. Available: www.ermt.net

[68] https://www.studocu.com/in/document/adithya-institute-of-technology/computer science/machine-learningr 17a0534/33563556

[69] S. Moustafa, M.Y. ElNainay, N. El Makky and M.S. Abougabal, "Software bug prediction using weighted majority voting techniques", Alexandria engineering journal, Vol. 57, No. 4, pp. 2763-2774, 2018. https://doi.org/10.1016/j.aej.2018.01.003

[70] Ö. F. Arar and K. Ayan, “Software defect prediction using cost-sensitive neural network,” *Applied Soft Computing Journal*, vol. 33, pp. 263–277, Apr. 2015, doi: 10.1016/j.asoc.2015.04.045.

[71] I. F. of E. Christ University (Bangalore and Institute of Electrical and Electronics Engineers, *2019 International Conference on Data Science and Communication (IconDSC) : Faculty of Engineering, CHRIST (Deemed to be University), Bangalore, 2019-03-01 to 2019-03-02.*

[72] S. Gupta and S. Kumar Gupta, “A Systematic Study of Duplicate Bug Report Detection.” [Online]. Available: www.ijacsa.thesai.org

[73] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, “Semantic Similarity Metrics for Evaluating Source Code Summarization,” in *IEEE International Conference on Program Comprehension*, IEEE Computer Society, 2022, pp. 36–47. doi: 10.1145/nnnnnnn.nnnnnnn.

[74] F. Barchi, E. Parisi, G. Urgese, E. Ficarra, and A. Acquaviva, “Exploration of Convolutional Neural Network models for source code classification,” *Eng Appl Artif Intell*, vol. 97, Jan. 2021, doi: 10.1016/j.engappai.2020.104075.

[75] H. Liu, Z. Xu, and Y. Zou, “Deep learning based feature envy detection,” in *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Association for Computing Machinery, Inc, Sep. 2018, pp. 385–396. doi: 10.1145/3238147.3238166.

[76] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, “Code smell detection by deep direct-learning and transfer-learning,” *Journal of Systems and Software*, vol. 176, Jun. 2021, doi: 10.1016/j.jss.2021.110936.

[77] L. Qiao, X. Li, Q. Umer, and P. Guo, “Deep learning based software defect prediction,” *Neurocomputing*, vol. 385, pp. 100–110, Apr. 2020, doi: 10.1016/j.neucom.2019.11.067.

[78] E. N. Akimova *et al.*, “A survey on software defect prediction using deep learning,” *Mathematics*, vol. 9, no. 11. MDPI AG, Jun. 01, 2021. doi: 10.3390/math9111180.

[79] M. A. Ramdhani, M. A. Ramdhani, D. S. adillah Maylawati, and T. Mantoro, “Indonesian news classification using convolutional neural network,” *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 19, no. 2, pp. 1000–1009, Aug. 2020, doi: 10.11591/ijeecs.v19.i2.pp1000-1009.

[80] C. Pan, M. Lu, B. Xu, and H. Gao, “An improved CNN model for within-project software defect prediction,” *Applied Sciences (Switzerland)*, vol. 9, no. 10, May 2019, doi: 10.3390/app9102138.

[81] S. I. Ayon, “Neural Network based Software Defect Prediction using Genetic Algorithm and Particle Swarm Optimization,” in *1st International Conference on Advances in Science, Engineering and Robotics Technology 2019, ICASERT 2019*, Institute of Electrical and Electronics Engineers Inc., May 2019. doi: 10.1109/ICASERT.2019.8934642.

[82] S. K. Pandey, R. B. Mishra, and A. K. Tripathi, “BPDET: An effective software bug prediction model using deep representation and ensemble learning techniques,” *Expert Syst Appl*, vol. 144, Apr. 2020, doi: 10.1016/j.eswa.2019.113085.

[83] Institute of Electrical and Electronics Engineers, *16th ACS/IEEE International Conference on Computer Systems and Applications AICCSA 2019 : 3 November to 7 November 2019, Al Ain University & Crowne Plaza, Abu Dhabi, UAE*.

[84] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, “Software Defect Prediction via Attention-Based Recurrent Neural Network,” *Sci Program*, vol. 2019, 2019, doi: 10.1155/2019/6230953.

[85] Z. Yang and H. Qian, “Automated parameter tuning of artificial neural networks for software defect prediction,” in *ACM International Conference Proceeding Series*, Association for Computing Machinery, Jun. 2018, pp. 203–209. doi: 10.1145/3239576.3239622.

[86] M. N. Uddin, B. Li, Z. Ali, P. Kefalas, I. Khan, and I. Zada, “Software defect prediction employing BiLSTM and BERT-based semantic feature,” *Soft comput*, vol. 26, no. 16, pp. 7877–7891, Aug. 2022, doi: 10.1007/s00500-022-06830-5.

[87] S. Feng, J. Keung, X. Yu, Y. Xiao, and M. Zhang, “Investigation on the stability of SMOTE-based oversampling techniques in software defect prediction,” *Inf Softw Technol*, vol. 139, Nov. 2021, doi: 10.1016/j.infsof.2021.106662.

[88] X. Li, J. Li, Y. Qu, and D. He, “Gear pitting fault diagnosis using integrated CNN and GRU network with both vibration and acoustic emission signals,” *Applied Sciences (Switzerland)*, vol. 9, no. 4, Feb. 2019, doi: 10.3390/app9040768.

[89] X. Bai, H. Zhou, and H. Yang, “An HVSM-based GRU approach to predict cross-version software defects,” *International Journal of Performability Engineering*, vol. 16, no. 6, pp. 979–990, Jun. 2020, doi: 10.23940/ijpe.20.06.p16.979990.

[90] S. M. Abd Elrahman and A. Abraham, “A Review of Class Imbalance Problem,” 2013. [Online]. Available: www.mirlabs.net/jnic/index.html

[91] L. Wang, M. Han, X. Li, N. Zhang, and H. Cheng, “Review of Classification Methods on Unbalanced Data Sets,” *IEEE Access*, vol. 9, pp. 64606–64628, 2021, doi: 10.1109/ACCESS.2021.3074243.

[92] F. Rodríguez-Torres, J. F. Martínez-Trinidad, and J. A. Carrasco-Ochoa, “An Oversampling Method for Class Imbalance Problems on Large Datasets,” *Applied Sciences (Switzerland)*, vol. 12, no. 7, Apr. 2022, doi: 10.3390/app12073424.

[93] C. Padurariu and M. E. Breaban, “Dealing with data imbalance in text classification,” in *Procedia Computer Science*, Elsevier B.V., 2019, pp. 736–745. doi: 10.1016/j.procs.2019.09.229.

[94] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, “A large empirical assessment of the role of data balancing in machine-learning-based code smell detection,” *Journal of Systems and Software*, vol. 169, Nov. 2020, doi: 10.1016/j.jss.2020.110693.

[95] E. AT, A. M, A.-M. F, and S. M, “Classification of Imbalance Data using Tomek Link (T-Link) Combined with Random Under-sampling (RUS) as a Data Reduction Method,” *Global Journal of Technology and Optimization*, vol. 01, no. S1, 2016, doi: 10.4172/2229-8711.s1111.

[96] T. T. Khuat and M. H. Le, "Evaluation of Sampling-Based Ensembles of Classifiers on Imbalanced Data for Software Defect Prediction Problems," *SN Comput Sci*, vol. 1, no. 2, Mar. 2020, doi: 10.1007/s42979-020-0119-4.

[97] N. M. Mqadi, N. Naicker, and T. Adeliyi, "Solving Misclassification of the Credit Card Imbalance Problem Using near Miss," *Math Probl Eng*, vol. 2021, 2021, doi: 10.1155/2021/7194728.

[98] IEEE Communications Society. Indonesia Chapter., Universitas Telkom., and Institute of Electrical and Electronics Engineers, *Proceedings, the 2020 IEEE International Conference on Industry 4.0, Artificial Intelligence, and Communications Technology : July 7-8, 2020, Bali, Indonesia.*

[99] E. F. Swana, W. Doorsamy, and P. Bokoro, "Tomek Link and SMOTE Approaches for Machine Fault Classification with an Imbalanced Dataset," *Sensors*, vol. 22, no. 9, May 2022, doi: 10.3390/s22093246.

[100] V. * Rajkumar and V. Venkatesh, "Hybrid Approach for Fault Prediction in Object-Oriented Systems," 2017.

[101] A. Iqbal *et al.*, "Performance Analysis of Machine Learning Techniques on Software Defect Prediction using NASA Datasets," 2019. [Online]. Available: www.ijacsa.thesai.org

[102] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy, "A public unified bug dataset for Java," in *ACM International Conference Proceeding Series*, Association for Computing Machinery, Oct. 2018, pp. 12–21. doi: 10.1145/3273934.3273936.

[103] A. B. Farid, E. M. Fathy, A. S. Eldin, and L. A. Abd-Elmegid, "Software defect prediction using hybrid model (CBIL) of convolutional neural network (CNN) and bidirectional long short-term memory (Bi-LSTM)," *PeerJ Comput Sci*, vol. 7, pp. 1–22, 2021, doi: 10.7717/peerj-cs.739.

[104] J. Deng, L. Lu, and S. Qiu, "Software defect prediction via LSTM," *IET Software*, vol. 14, no. 4, pp. 443–450, Aug. 2020, doi: 10.1049/iet-sen.2019.0149.

[105] E. Tempero *et al.*, "The Qualitas Corpus: A curated collection of Java code for empirical studies," in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2010, pp. 336–345. doi: 10.1109/APSEC.2010.46.

[106] D. L. Miholca, G. Czibula, and I. G. Czibula, "A novel approach for software defect prediction through hybridizing gradual relational association rules with artificial neural networks," *Inf Sci (N Y)*, vol. 441, pp. 152–170, May 2018, doi: 10.1016/j.ins.2018.02.027.

[107] L. Zhao, Z. Shang, L. Zhao, T. Zhang, and Y. Y. Tang, "Software defect prediction via cost-sensitive Siamese parallel fully-connected neural networks," *Neurocomputing*, vol. 352, pp. 64–74, Aug. 2019, doi: 10.1016/j.neucom.2019.03.076.

[108] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "A novel approach for code smell detection: An empirical study," *IEEE Access*, vol. 9, pp. 162869–162883, 2021, doi: 10.1109/ACCESS.2021.3133810.

[109] M. Z. Khan, "Hybrid ensemble learning technique for software defect prediction," *International Journal of Modern Education and Computer Science*, vol. 12, no. 1, pp. 1–10, 2020, doi: 10.5815/ijmecs.2020.01.01.

[110] Z. Li, X. Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *IET Software*, vol. 12, no. 3. Institution of Engineering and Technology, pp. 161–175, Jun. 01, 2018. doi: 10.1049/iet-sen.2017.0148.

[111] M. A. Ihsan Aquil, "Predicting Software Defects using Machine Learning Techniques," *International Journal of Advanced Trends in Computer Science and Engineering*, vol. 9, no. 4, pp. 6609–6616, Aug. 2020, doi: 10.30534/ijatcse/2020/352942020.

[112] N. A. A. Khleel and K. Nehéz, "A new approach to software defect prediction based on convolutional neural network and bidirectional long short-term memory," *Production Systems and Information Engineering*, vol. 10, no. 3, pp. 1–15, 2022, doi: 10.32968/psaie.2022.3.1.

[113] K. Boyd, K. H. Eng, and C. David Page, "Area Under the Precision-Recall Curve: Point Estimates and Confidence Intervals." [Online]. Available: http://link.springer.com/chapter/10.1007%

[114] S. Singh and A. Professor, "Software Bug Prediction using Machine Learning Approach," *International Research Journal of Engineering and Technology*, p. 4968, 2008, [Online]. Available: www.irjet.net

[115] H. Tong, S. Wang, and G. Li, "Credibility based imbalance boosting method for software defect proneness prediction," *Applied Sciences (Switzerland)*, vol. 10, no. 22, pp. 1–29, Nov. 2020, doi: 10.3390/app10228059.

[116] H. S. Munir, S. Ren, M. Mustafa, C. N. Siddique, and S. Qayyum, "Attention based GRU-LSTM for software defect prediction," *PLoS One*, vol. 16, no. 3 March, Mar. 2021, doi: 10.1371/journal.pone.0247444.

[117] R. Ferenc, D. Bán, T. Grósz, and T. Gyimóthy, "Deep learning in static, metric-based bug prediction," *Array*, vol. 6, p. 100021, Jul. 2020, doi: 10.1016/j.array.2020.100021.

[118] D. Cruz, A. Santana, and E. Figueiredo, "Detecting bad smells with machine learning algorithms: An empirical study," in *Proceedings - 2020 IEEE/ACM International Conference on Technical Debt, TechDebt 2020*, Association for Computing Machinery, Inc, Jun. 2020, pp. 31–40. doi: 10.1145/3387906.3388618.

[119] M. Hozano, N. Antunes, B. Fonseca, and E. Costa, "Evaluating the accuracy of machine learning algorithms on detecting code smells for different developers," in *ICEIS 2017 - Proceedings of the 19th International Conference on Enterprise Information Systems*, SciTePress, 2017, pp. 474–482. doi: 10.5220/0006338804740482.

[120] S. Jain and A. Saha, "Rank-based univariate feature selection methods on machine learning classifiers for code smell detection," *Evol Intell*, vol. 15, no. 1, pp. 609–638, Mar. 2022, doi: 10.1007/s12065-020-00536-z.