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

Table of Contents

1 Introduction 1

2 Literature Review and Theoretical Background 1

2.1 Software Bugs 1

2.2 Code Smells 2

2.3 Software Metrics 3

3 Artificial Intelligence Techniques 3

3.1 Machine Learning (ML) 4

3.2 Artificial Neural Networks (ANNs) 5

4 Data Imbalance and Data-Balancing Methods 6

4.1 Data Imbalance 6

4.2 Data-Balancing Methods 6

4.2.1 Data Sampling (Resampling) Methods 7

5 Proposed Methodology and Implementation 8

5.1 Experimental Design 9

5.1.1 Proposed Approaches 9

5.1.2 The Public Benchmark Datasets Used in This Research 9

5.1.3 Data Pre-processing 10

5.1.4 Features Selection 11

5.1.5 Balancing Data sets 12

5.1.6 Models Building and Evaluation 12

6 Experimental Results and Discussion 13

6.1 Experimental Results and Discussion of Software Bugs Prediction (SBP) 13

6.1.1 ML Techniques in SBP 13

6.1.2 LSTM and GRU with Undersampling Methods in SBP 14

6.1.3 Bi-LSTM with Oversampling Methods in Software Defect Prediction (SDP) 15

6.1.4 CNN and GRU with Hybrid (combined)-Sampling Methods in SDP 18

6.2 Experimental Results and Discussion of Code Smells Detection 19

6.2.1 ML techniques with Oversampling Methods in Code Smells Detection 19

6.2.2 A Convolutional Neural Network (CNN) with Oversampling Methods 22

6.2.3 Bi-LSTM and GRU with Under and Oversampling Methods in Code Smells Detection 23

6.3 Summary 25

7 Thesis Summary 26

Author's Publication 28

Publications Related to the Dissertation 28

Other Publications Journal Articles and Conference Proceeding 29

References 30

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 [2]. Detecting and predicting these issues early in the software development life cycle can save substantial time, effort, and resources [3], [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 [5], [6]. 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. The most popular AI techniques used for the prediction of software bugs and code smells are Machine Learning (ML) techniques [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 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. 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].

2 Literature Review and Theoretical Background

2.1 Software Bugs

Due to the expansion in the scale of software projects and the increase in complexity, Software Bug Prediction (SBP) has become the focus of attention to increase software quality. 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. 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.[7]. 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. 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. 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. 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. 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"[11].

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. Code smells may lead to future degradation in software projects making software hard to evolve and maintain, and it can effectively indicate whether source code should be refactored. 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. 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. Many approaches have been presented by the authors for uncovering the smells from the software systems[12]. 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[13].

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 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. 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[5]. 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. 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. 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. Halstead also considered that the software metrics fall into three groups: base measures, derived measures, and line of code measures [3], [6].

3 Artificial Intelligence Techniques

The field of Artificial intelligence (AI) is witnessing a recent upsurge in research, tools development, and deployment of applications. 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. 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[14].

3.1 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. 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. ML algorithms build models from training examples, which are then used to make predictions when faced with new examples. ML techniques can be categorized into supervised, unsupervised, and reinforcement. 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[8].

3.1.1 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. 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[9]. 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. The most commonly supervised ML methods include concept learning, classification, rule learning, instance-based learning, Bayesian learning, linear regression, neural network, SVM, etc.[8].

3.1.2 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[7]. 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. 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[10].

3.1.3 Reinforcement learning

Reinforcement learning is somewhere between supervised and unsupervised learning. Reinforcement learning is applied when the task at hand is to make a sequence of decisions toward a final reward[10]. 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. 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. Examples include learning agents to play computer games or performing robotics tasks with end goals[9].

3.2 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[7]. 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. 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. 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 [14].

3.2.1 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 [7], [9]. 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. 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 [10], [14].

3.2.2 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 [9]. RNN is widely used to solve many different problems, such as pattern recognition, identification, classification, vision, speech, control systems, etc. 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. 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. [7], [10].

4 Data Imbalance and Data-Balancing 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. 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 [7]. 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. 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. 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[13].

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 [7]. 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[13]. 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. 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. 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) [15].

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)[13]. 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. There are three main categories of data sampling techniques that are: Oversampling Methods, Undersampling Methods, and Hybrid (Combined-Sampling Methods) [7], [15].

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. 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, 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[13].
- 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 [15].

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. Oversampling methods are more effective than Undersampling methods in prediction accuracy [7]. There are many Oversampling methods

such as Random Oversampling, Synthetic Minority Oversampling Technique (SMOTE), etc. [13].

- 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[15].
- 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. It is the most widely used and referenced method among the Oversampling methods. 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[13].

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. 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[15].

- 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[13].

5 Proposed Methodology and Implementation

This section 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 1.

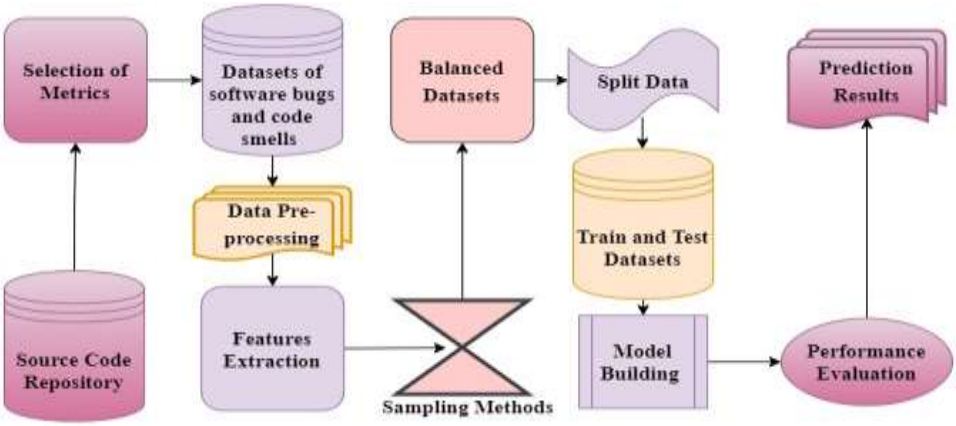

Figure 1 The architecture of the methodology followed in the dissertation

5.1 Experimental Design

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

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 [16]. 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[17]. 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. 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 [18]. 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.

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. The Qualitas Corpus (QC) systems were collected by Tempero et al. 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].

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[7]. 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. 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[19].

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. 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 [7]. 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 [19]. 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. 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 [7].

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]. 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[14]. 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.

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 [7]. 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) [19].

6 Experimental Results and Discussion

6.1 Experimental Results and Discussion of Software Bugs Prediction (SBP)

6.1.1 ML Techniques in SBP

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, and f-measure). The performance of the prediction models is reported in Tables 1 to 4.

Tables 1 to 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 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 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 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 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

6.1.2 LSTM and GRU with Undersampling Methods in SBP

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 5 and 6.

Table 5 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 5 Performance measures for the proposed models over class level metrics dataset

Original Dataset								
	Performance Measures							
Proposed Models	**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								
	Performance Measures							
Proposed Models	**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 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 Performance measures for the proposed models over file level metrics dataset

Original Dataset								
	Performance Measures							
Proposed Models	**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								
	Performance Measures							
Proposed Models	**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**

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

The aim 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 7 to 9.

According to Table 7: 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 7 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 8: 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 8 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 9: 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 9 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**

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

The target 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 10 to 13.

Table 10 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 11 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 12 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 13 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**

6.2 Experimental Results and Discussion of Code Smells Detection

6.2.1 ML techniques with Oversampling Methods in Code Smells Detection

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 14 to 17.

Tables 14 to 17 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 14 Evaluation Results for the Class-Level Dataset: God class_ original and balanced datasets

Original datasets						
	Performance measurement					
ML Models	**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						
	Performance measurement					
ML Models	**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 15 Evaluation Results for the Class-Level Dataset: Data class_ original and balanced datasets

Original datasets						
	Performance measurement					
ML Models	**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						
	Performance measurement					
ML Models	**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 16 Evaluation Results for the Method-Level Dataset: Long method_ original and balanced datasets

Original datasets						
	Performance measurement					
ML Models	**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						
	Performance measurement					
ML Models	**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 17 Evaluation Results for the Method-Level Dataset: Feature envy_ original and balanced datasets

Original datasets						
	Performance measurement					
ML Models	**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						
	Performance measurement					
ML Models	**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**

6.2.2 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).

Tables 18 and 19 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 18 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 19 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**

6.2.3 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 20 to 22.

Table 20 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 20 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 21 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 21 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 22 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 22 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**

6.3 Summary

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. We concluded that the combined data-balancing methods with ML techniques significantly enhance the accuracy of predicting software bugs and code smells. We observe that the incorporation of appropriate data-balancing methods and ML techniques not only enhances the model's ability to predict software bugs and code smells accurately but also mitigates the bias towards the majority class, resulting in a more balanced performance across different classes of software bugs and 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 predicting software bugs and code smells. By employing these methods, developers can enhance their ability to identify and address code quality issues, thereby improving software maintainability.

7 Thesis Summary

The new scientific results of the research presented in this work are as follows:

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].

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

[17] 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.

[18] 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

[19] 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.

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