ISSN 2442-6571

Detection of code smells using machine learning techniques combined with data-balancing methods

Nasraldeen Alnor Adam Khleel [a,1,*], Károly Nehéz [a,2]

[a] Department of Information Engineering, University of Miskolc, Miskolc, H-3515, Hungary
[1] nasr.alnor@uni-miskolc.hu; [2] aitnehez@uni-miskolc.hu
* corresponding author

ARTICLE INFO

Article history
Received January 1, 2023
Revised April 5, 2023
Accepted April 16, 2023
Available online October 15, 2023

Keywords
Code smells
Software metrics
Machine learning techniques
Class Imbalance
Data balancing methods

ABSTRACT

Code smells are prevalent issues in software design that arise when implementation or design principles are violated. These issues manifest as symptoms or anomalies in the source code. Timely identification of code smells plays a crucial role in enhancing software quality and facilitating software maintenance. Previous studies have shown that code smell detection can be accomplished through the utilization of machine learning (ML) methods. However, despite their increasing popularity, research suggests that the suitability of these methods are not always appropriate due to the problem of imbalanced data. Consequently, the effectiveness of ML models may be negatively affected. This study aims to propose a novel method for detecting code smells by employing five ML algorithms, namely decision tree (DT), k-nearest neighbors (K-NN), support vector machine (SVM), XGboost (XGB), and multi-layer perceptron (MLP). Additionally, to tackle the challenge of imbalanced data, the proposed method incorporates the random oversampling technique. Experiments were conducted in this study using four datasets that encompassed code smells, specifically god-class, data-class, long-method, and feature-envy. The experimental outcomes were evaluated and compared using various performance metrics. Upon comparing the outcomes of our models on both the balanced and original datasets, we found that the XGB model achieved the highest accuracy of 100% for detecting the data class and long method on the original datasets. In contrast, the highest accuracy of 100% was obtained for the data class and long method using DT, SVM, and XGB models on the balanced datasets. According to the empirical findings, there is significant promise in using ML techniques for the accurate prediction of code smells.

1. Introduction

Code smells indicate design issues that violate basic design principles such as hierarchy encapsulation, abstraction, and others, potentially affecting software quality [1], [2]. Detecting code smells is crucial for guiding the subsequent refactoring process to improve software quality and reduce software failure risk [3], [4]. Code smells usually appear during the design or coding phase due to developers' activities in emergencies, inadequate design, or coding solutions [5], [6]. Table 1 lists the four specific code smells that we have investigated.

Software metrics are crucial in measuring and enhancing software quality and are utilized to characterize software engineering products [7]. They have diverse applications, including bug identification, test complexity prediction, code smell detection, and clone prediction. Object-oriented metrics are the most commonly used software metrics [8], [9].

Table 1. Lists the four specific code smells that we have investigated [9].

Code smells	Description	Affected entity
God _Class	Refers to classes that have many members and implement different behaviors.	Class
Data _Class	Refers to classes that contain data only.	Class
Long _Method	Refers to the too-long method.	Method
Feature _Envy	Refers to a method that displays a greater interest in the properties of other classes than those belonging to its class.	Method

Many techniques and tools have been devised for detecting code smells, encompassing manual, automatic, and metrics-based approaches [2], [10]. However, most of these methods employ a heuristic two-step approach, which involves calculating metrics first and then applying threshold values to differentiate between smelly and non-smelly classes [6]. The dissimilarities among these strategies lie in the algorithms employed, the subjective interpretation involved, the lack of consensus among detectors, and the dependence on thresholds [11]. Researchers have recently turned to ML algorithms to mitigate these limitations in code smell detection to avoid thresholds and reduce the incidence of false positives in detection tools [12].

ML models are mathematical techniques that employ historical data to automatically identify intricate patterns and make informed and intelligent decisions [3]. Supervised ML techniques are commonly used for code smells detection [13], [14]. With supervised classification algorithms, the machine can acquire knowledge of the associations between instances and decision labels [10], [15].

Studies on code smells detection have recently gained more attention, and scientific researchers have presented many studies for code smells detection using ML models. For example, Mhawish and Gupta [1] presented an approach for predicting code smells using ML techniques and software metrics. The authors utilized datasets obtained from Fontana et al. and their experimental results showed that the accurate prediction of code smells can be significantly facilitated by employing ML techniques. Fabiano Pecorelli et al. [2] examined five distinct data balancing methods to alleviate issues of data imbalance and gauge their effects on ML algorithms in code smell detection. During the experiment, five datasets on code smells were utilized. The findings indicate that ML models utilizing the synthetic minority oversampling technique exhibited the most promising performance. This technique effectively addressed the problem of class imbalance. Fontana *et al.* [9] presented an approach for identifying code smells that involves the use of various ML techniques. The results indicate that all techniques performed satisfactorily, however, the imbalanced data adversely affected the performance of certain models. Cruz *et al.* [16] conducted an assessment of seven ML algorithms to identify four distinct types of code smells, while also analyzing the influence of software metrics on the detection of code smells. The experimental results found that ML algorithms can perform well in detecting bad code smells, and metrics play a fundamental role in detecting bad code smells. Martins *et al.* [17] conducted an empirical study to predict classes that are susceptible to change using eight ML techniques. In their study, three distinct training scenarios were involved, which included object-oriented metrics, code smells, and a fusion of both. The experiments were conducted on a dataset of 32 code odor types and eight object-oriented metrics. The experimental results found that some ML algorithms presented the best results based on the training scenario of a combination of code smells and object-oriented metrics. Hozano *et al.* [18] evaluated and compared the effectiveness of six ML algorithms in detecting four different code smells across a sample of 40 developers. The findings revealed that the ML algorithms performed poorly for the participating developers, indicating their susceptibility to the type of smells and the individual developer. These algorithms were unable to learn effectively from a limited training set. Sharma *et al.* [19] presented a method for code smells detection based on convolution neural networks and recurrent neural network models. The experiment results showed that detecting code smells is feasible using deep learning methods. Dewangan *et al.* [20] proposed an approach based on six ML algorithms to predict code smells based on four datasets obtained from 74 open-source systems. The proposed approach's effectiveness was assessed using various performance metrics, and two feature selection methods were implemented to improve the accuracy of the predictions. The experimental results showed that their approach achieved high prediction accuracy. Jain and Saha [21] proposed a method for code smell detection based on several

ML models. The method was evaluated based on different performance metrics. The experimental results demonstrate that boosted decision trees and Naive Bayes models yielded superior performance compared to other models, following dimensionality reduction.

Our analysis of prior research on code smells detection revealed that most proposed methods overlook the issue of class imbalance. However, studies that addressed this problem and implemented data balancing methods [2], [22] emphasized such methods' critical and essential role in code smells detection.

Data imbalance in a training data set, where classes are unevenly distributed, hinders the efficiency of ML algorithms and biases their performance towards the majority class [3]. That leads to unbalanced false-positive and false-negative results, making data imbalance the biggest problem for ML algorithms [22]. This study selects imbalanced datasets extracted from 74 open-source systems [9]. Consequently, there is a growing impetus to employ data balancing methods and develop unbiased classifiers that operate effectively on imbalanced code smells datasets.

To our knowledge, a few studies have applied ML combined with sampling techniques for code smells detection. To address these gaps, our work offers a novel method that aims to achieve the following key objectives and contributions:

- The present study introduces a novel method that combines machine learning (ML) with a random oversampling technique to effectively detect code smells.
- This study evaluates the efficiency of the proposed method utilizing various performance measures and compares it with the currently employed methods for detecting code smells.
- We show that the performance of ML models in code smells detection can be significantly improved when balancing the data set by applying data-balancing methods.

The paper is structured in the following manner: Section 2 specifies the research method. Section 3 outlines the results and corresponding discussions, while the final Section (Section 4) presents the conclusion.

2. Method

Our study proposes a method for training and testing code smell detection models, which utilizes high-performance supervised machine learning algorithms in combination with a random oversampling technique. Fig. 1 illustrates the proposed research process for detecting code smells. The following sections describe the steps taken in this study, which encompass dataset description, data pre-processing, feature selection, dataset balancing, classification algorithms, model building, and evaluation.

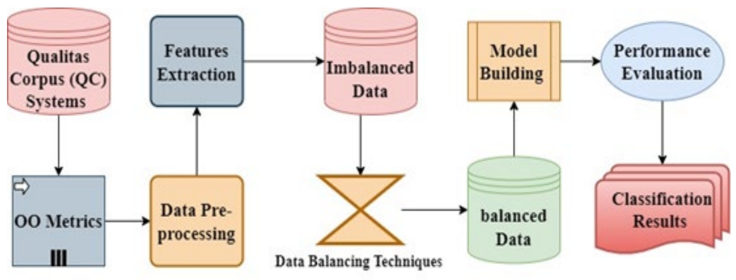

Fig. 1. Shows the overview of the proposed research process for detecting code smells.

2.1. Dataset Description

To perform the analysis and experiments, our method was implemented using the datasets proposed by Fontana *et al.* [9], which include 74 open-source systems of varying sizes and domains sourced from Qualitas Corpus (QC) [23], as detailed in Table 2. The justification for selecting these datasets is that the 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. In QC systems, metrics are chosen for both class and method levels. The metrics chosen comprise a standardized set of metrics that address various aspects of the code, such as size, cohesion, encapsulation, etc. [9]. The computed metrics for all 74 systems of the QC are displayed in Table 3.

Table 2. A summary of the QC systems [9].

The count of systems	Code Lines	The count of packages	The count of classes
74	6,785,568	3420	51,826

Table 3. Metrics computed on all 74 systems of the QC [9].

Size	Complexity	Cohesion	Coupling	Encapsulation	Inheritance
LOCNAMM*	WMC	LCOM5	ATFD	NOAM	NOII
NOM	CYCLO	TCC	FANOUT	NOPA	NOI
LOC	WOC		CINT	LAA	NIM
NOA	AMWNAMM*		RFC		NMO
NOMNAMM*	NOP		CDISP		NOC
NOCS	MAXNESTING		CFNAMM*		DIT
NOPK	WMCNAMM*		FDP		
	CLNAMM		CBO		
	AMW		MeMCL§		
	NOLV		MaMCL§		
	ATLD*		NMCS§		
	NOAV		CM		
			CC		

2.2. Data Pre-processing and Features Selection

Before constructing the model, it is essential to carry out pre-processing of the collected data. To ensure the production of an optimal model, careful attention must be paid to the quality of the data [24]. Data pre-processing refers to a collection of procedures utilized to enhance data quality before constructing a model. Its primary objectives are the removal of noise and extraneous outliers, managing missing values, converting feature types, and more [10], [11], [25]. Selecting the most informative features from a list of features through suitable methods is a crucial step commonly referred to as Feature Selection (FS). [26]–[28]. FS aims to identify the most relevant features for the target class from a high-dimensional feature set and eliminate redundant and uncorrelated features [1], [17], [21], [29]. There are three distinct categories of FS methods, which are wrapper methods, embedded methods, and filter methods, each method has rules for selecting the most relevant features as independent variables for training ML models [29]. In this study, our models were based on embedded methods because these methods fit ML models.

2.3. Class Imbalance and Sampling Techniques

Class imbalance is a common issue in code smells detection, wherein one class has significantly fewer examples than the others. That is particularly relevant since code smells datasets often consist of a small number of smelly instances and many non-smelly ones [3]. Therefore, the class imbalance problem can often lead to misclassifying cases in the minority class [30]. To address this problem, various techniques have been proposed, including data sampling methods, boosting-based ensemble methods, bagging-based ensemble methods, cost-sensitive learning approaches, and other similar approaches [2]. the dataset used for code smells detection in this study is significantly imbalanced [15]. Specifically, the initial 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, namely feature envy (with 140 smelly instances and 280 non-smelly instances) and long method (with 141 smelly instances and 279 non-smelly instances). We address the class imbalance issue by enhancing the original datasets to make the data more realistic. We mitigate the class imbalance problem by using the random oversampling technique, which involves randomly selecting examples to increase the minority class [2], [17]. Fig. 2 illustrates the learning instances distribution in all datasets.

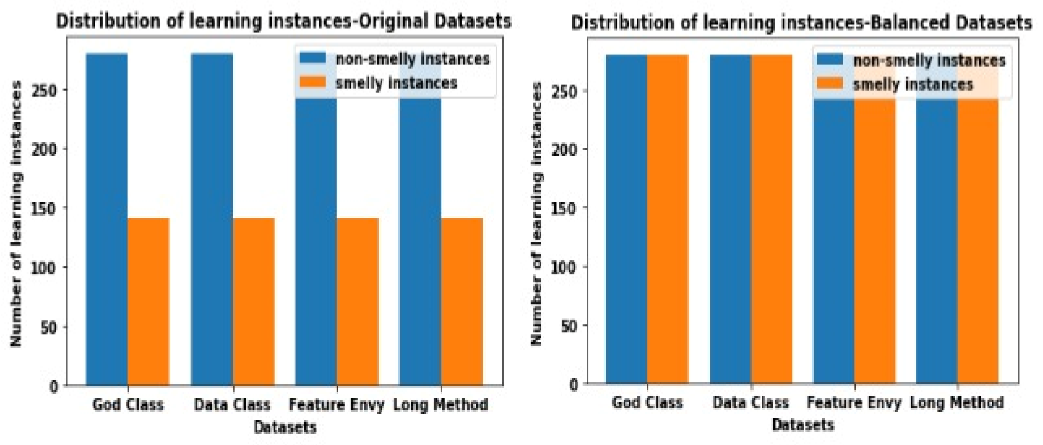

Fig. 2. illustrates the learning instances distribution in all datasets.

2.4. Classification Algorithms

This section briefly describes the classification algorithms used to detect and classify code smells in our study.

2.4.1. DT

DT is a supervised ML algorithm utilized for carrying out both regression and classification tasks [9]. DTs function by segregating instances based on feature values and branching them out. In an ID3 decision tree, all features are initially assigned as root nodes. Subsequently, the features are separated by computing their Entropy, which measures data homogeneity. Entropy values fall within the range of 0 to 1[12], [13]. Mathematically, entropy for a single attribute is represented as:

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

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

2.4.2. K-NN

K-NN is a basic supervised ML algorithm that operates by examining K neighboring objects and selecting the most commonly occurring class or calculating the distance between them [28]. Additionally, it is a lazy-learning technique that categorizes elements according to their spatial arrangement on a hyperplane. The algorithm necessitates the selection of k closest points. Therefore, the first stage involves determining the distance between the input data point and other points in our training data [29], [31]. The distance between these two points can be calculated using:

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

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

2.4.3. SVM

SVM is a widely-used and regularized machine learning algorithm employed mainly for classification and regression purposes. SVMs leverage a margin on both sides of a hyperplane to separate two features, and they optimize the hyperplane to maximize the margin between features [2], [29]. The general form of the SVM function is defined as:

$$F(x) = W * Q(x) + b \tag{3}$$

In this context, the weight vector is denoted by w, the input vector is represented by x, and the hyperplane equations' intercept and bias terms are indicated by b.

2.4.4. XGB

XGB is a robust ML algorithm that has been recently introduced. It is grounded in the principle of gradient boosting and utilizes parallel tree boosting to predict the target through the consolidated results of numerous weak models. XGB delivers exceptional speed and accuracy. The formula for the XGB model is given as:

$$Y_i \ = w_q(x_i) = \sum_{t=1}^{T} f_t\,(x_i) \tag{4}$$

Where $f_t(x_i) \ = \ w_q(x)$ represents the domain of classification trees, while $w_q(x)$ represents the score of a particular sample, x represents the predicted value generated by the model. In addition, q signifies the structure of each tree, T refers to the total number of trees, and each f_t corresponds to a different tree structure q with its corresponding leaf weight w.

2.4.5. MLP

MLP is an artificial neural network composed of multiple layers of interconnected perceptrons, specifically designed to handle intricate data inputs and execute diverse tasks such as regression or classification. The network employs nodes with specific weights to create connections between these layers. The backpropagation algorithm is used to train the model in the MLP network [7]. The formula of the MLP model is as follows:

$$Y_i \ = f_i(\ \sum_{j=1}^{n} X_j\, W_{ij} \ + \ b_i) \tag{5}$$

The given context states that the output is denoted byY_i , while n represents the overall number of inputs that are provided to the neuron. Additionally, X_j represents the input to the network, The weights of the connections between input and output nodes are denoted byW_{ij} , the bias term is represented byb_i, and the transfer function is symbolized byf_i.

2.5. Models Building and Evaluation

The proposed models were built and evaluated by utilizing 80% of the dataset for training and keeping the remaining 20% for validation. Table 4 outlines the various parameters used for creating each model independently. We assess the effectiveness of our proposed models by utilizing standard evaluation metrics, namely the confusion matrix, which includes measures such as (accuracy, precision, recall, and f- measure), MCC, and AUC. MCC is a widely adopted metric for model assessment, which captures the variation between predicted and actual values through true and false positives and negatives. AUC is a visual depiction of classifier efficacy that plots the true positive rate versus the false positive rate at varying classification thresholds. A confusion matrix is a table that assists in evaluating the performance of a classification model by comparing the predicted class labels to the actual class labels of a dataset as illustrated in Table 5.

Table 4. Shows the parameters setting for the models

Models	Parameters setting
DT	No passing parameters (default parameters)
K-NN	N_neighbors = 3
SVM	Probability = True, kernel = 'linear'
XGB	Max_depth=3, n_estimators=100, n_jobs=2, objectvie='binary:logistic', learning_rate=0.01, Subsample=0.7, colsample_bytree=0.8
MLP	Hidden_layer_sizes=(10,5), max_iter=1000

Table 5. Shows the confusion matrix

Predicted class	Actual class	
	Yes	*No*
Yes	TP	FN
No	FP	TN

$$\text{Accuracy} = \frac{(\text{TP+TN})}{(\text{TP+FP+FN+TN})} \tag{6}$$

$$\text{Precision} = \frac{\text{TP}}{(\text{TP+FP})} \tag{7}$$

$$\text{Recall} = \frac{\text{TP}}{(\text{TP + FN})} \tag{8}$$

$$\text{F} - \text{Measure} = \frac{(2 * \text{Recall} * \text{Precision})}{(\text{Recall + Precision})} \tag{9}$$

$$\text{MCC} = \text{TP} * \text{TN} - \text{FP} * \text{FN} \,/ \sqrt{(\text{TP + FP}) * (\text{TP + FN}) * (\text{TN + FP}) * (\text{TN + FN})} \tag{10}$$

$$\text{AUC} = \frac{\sum_{ins_i \in Positive\ Class} \text{rank}(ins_i) - \frac{\text{M(M+1)}}{2}}{\text{M . N}} \tag{11}$$

In this given scenario, the sum of ranks for all positive samples is represented by$\sum_{ins_i \in Positive\ Class} \text{rank}(ins_i)$, while the number of positive examples is denoted by "M," and the number of negative examples is represented by "N".

3. Results and Discussion

The experimental setup was implemented in Python, and the training and validation datasets were obtained from the same project. To ensure reliable performance evaluation, the proposed models were trained and tested on large datasets with over 6,785,568 lines of source code. Table 6 to Table 9, and Fig. 3 to Fig. 7 show the results.

Table 6. The results for the class-level dataset: god class _ original and balanced datasets

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

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.

Table 7. The results for the class-level dataset: data class_ original and balanced datasets

Machine Learning Models	Accuracy	Precision	Recall	F- measure	MCC	AUC
Original datasets						
	Performance measurement					
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					
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**

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.

Table 8. The results for the method-level dataset: long method_ original and balanced datasets

Machine Learning Models	Accuracy	Precision	Recall	F- measure	MCC	AUC
Original datasets						
	Performance measurement					
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					
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**

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.

Table 9. The results for the method-level dataset: feature envy_ original and balanced datasets

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

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).

Fig. 3 shows the best accuracy values of the models for all considered code smells on the original and balanced datasets. The best accuracy on the original datasets (god class) is 98% obtained by the XGB model, while the best accuracy on the balanced datasets (god class) is 98% obtained by the DT model. The best accuracy on the original datasets (data class) is 100% which the XGB model gets, while the best accuracy on the balanced datasets (data class) is 100% obtained by the DT and XGB models. The best accuracy on the original datasets (long method) is 100% which the XGB model gets, while the best accuracy on the balanced datasets (long method) is 100% which is obtained by the DT, SVM, and XGB models. The best accuracy on the original datasets (feature envy) is 95% which the XGB model gets. The best accuracy on the balanced datasets (feature envy) is 98%, obtained by the DT and XGB models.

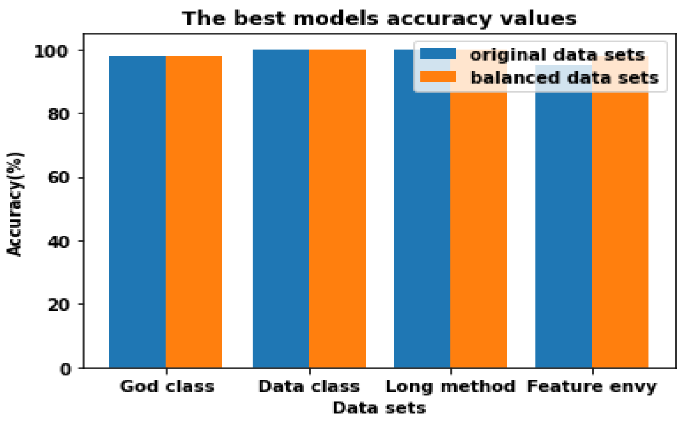

Fig. 3. The best models' accuracy values on original and balanced data sets.

Fig. 4 exhibits box plots that display the averages of several performance measures, including accuracy, precision, recall, f-measure, MCC, and AUC based on the original datasets. The overall average performance of all models is 0.93, 0.96, 0.88, 0.92, 0.86, and 0.96, respectively, for the god class. Similarly, for the data class, the overall average performance of all models is 0.96, 0.91, 0.95, 0.93, 0.90, and 0.98, respectively. In the context of the long method, the overall average of all models is 0.96, 0.95, 0.93, 0.94, 0.91, and 0.97, respectively. Lastly, for feature envy, the overall average performance of all models is 0.90, 0.85, 0.83, 0.84, 0.77, and 0.92, respectively.

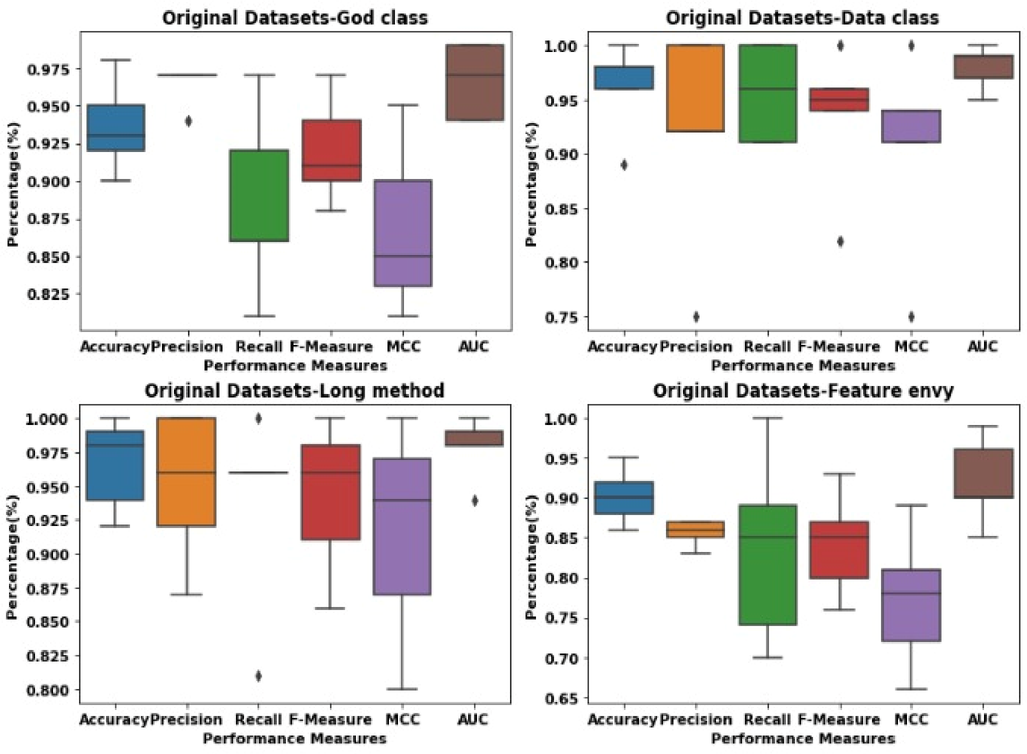

Fig. 4. Box Plots represent the models' performance measures on all considered code smells_ original datasets.

Fig. 5 exhibits box plots that display the averages of several performance measures, including accuracy, precision, recall, f-measure, MCC, and AUC based on the balanced datasets. The overall average performance of all models is 0.96, 0.96, 0.98, 0.97, 0.93, and 0.98, respectively, for the god class. Similarly, for the data class, the overall average performance of all models is 0.98, 0.97, 0.99, 0.98, 0.96, and 0.99, respectively. In the context of the long method, the overall average of all models is 0.98, 0.97, 0.99, 0.98, 0.97, and 0.99, respectively. Lastly, for feature envy, the overall average performance of all models is 0.95, 0.94, 0.98, 0.96, 0.91, and 0.96, respectively.

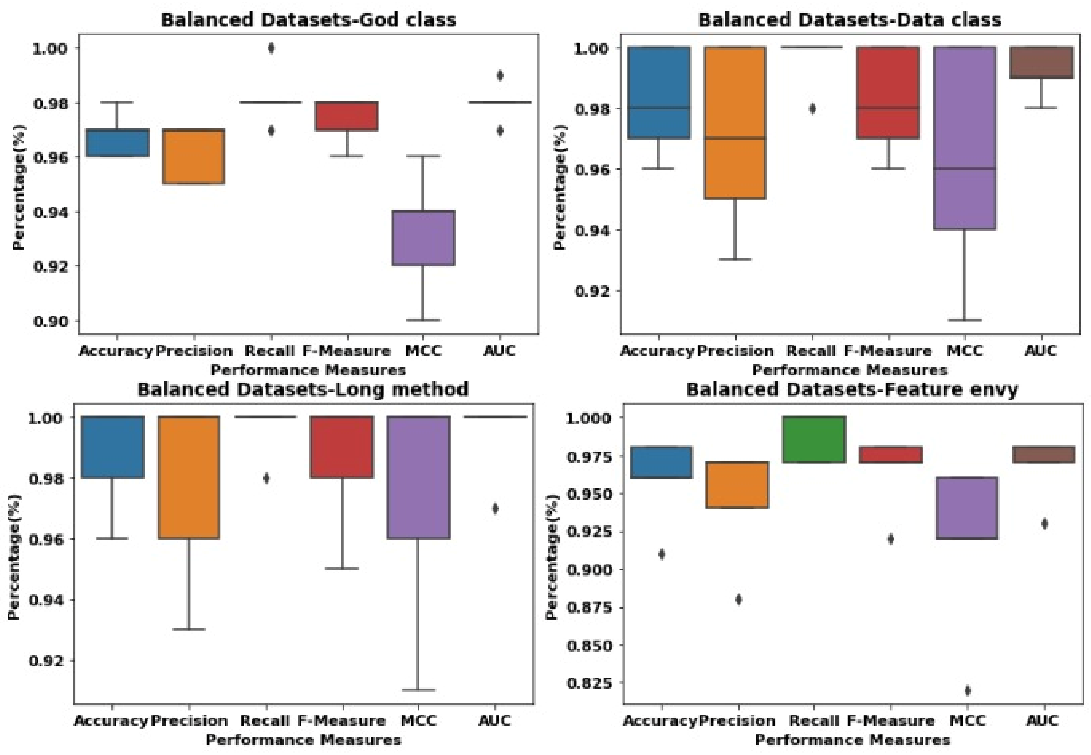

Fig. 5. Box Plots represent the models' performance measures on all considered code smells_ balanced datasets.

Fig. 6 shows the AUC of the models for all considered code smells on the original datasets; the highest AUC on the original datasets (god class) is 99%, obtained by XGB and MLP models.

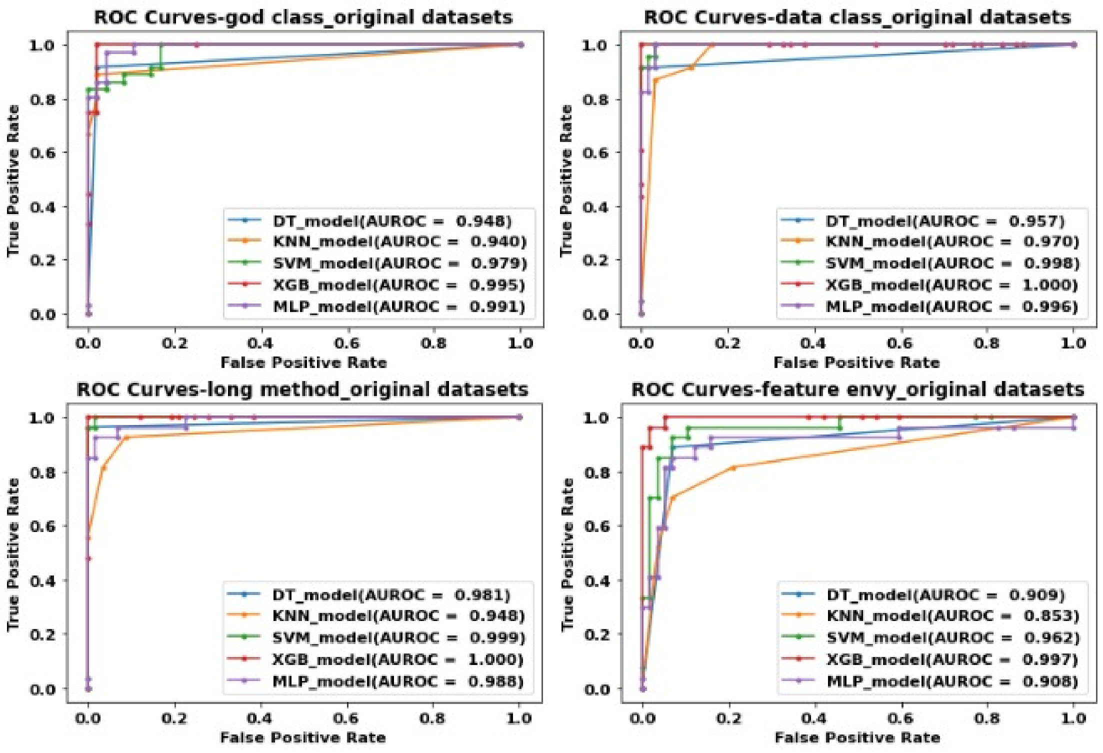

Fig. 6. The ROC curves obtained by the models on all considered code smells_ original datasets.

In comparison, the lowest AUC is 94%, obtained by DT and K-NN models. The highest AUC on the original datasets (data class) is 100% obtained by the XGB model, while the lowest AUC is 95% obtained by the DT model. The highest AUC on the original datasets (long method) is 100% obtained by the XGB model, while the lowest AUC is 94% obtained by the K-NN model. The highest AUC on the original datasets (feature envy) is 99%, obtained by the XGB model, while the lowest AUC is 85%, obtained by the K-NN model.

Fig. 7 shows the AUC of the models for all considered code smells on the balanced datasets, the highest AUC on the balanced datasets (god class) is 99%, obtained by the SVM model, while the lowest AUC is 97%, and the K-NN model gets. The highest AUC on the balanced datasets (data class) is 100% obtained by DT and XGB models, while the lowest AUC is 98% obtained by the K-NN model.

The highest AUC on the balanced datasets (long method) is 100% acquired by DT, SVM, XGB, and MLP models, while the lowest AUC is 97%, which the K-NN model obtains. The highest AUC on the balanced datasets (feature envy) is 99%, obtained by DT, XGB, and MLP models, while the lowest AUC is 93% which the K-NN model gets.

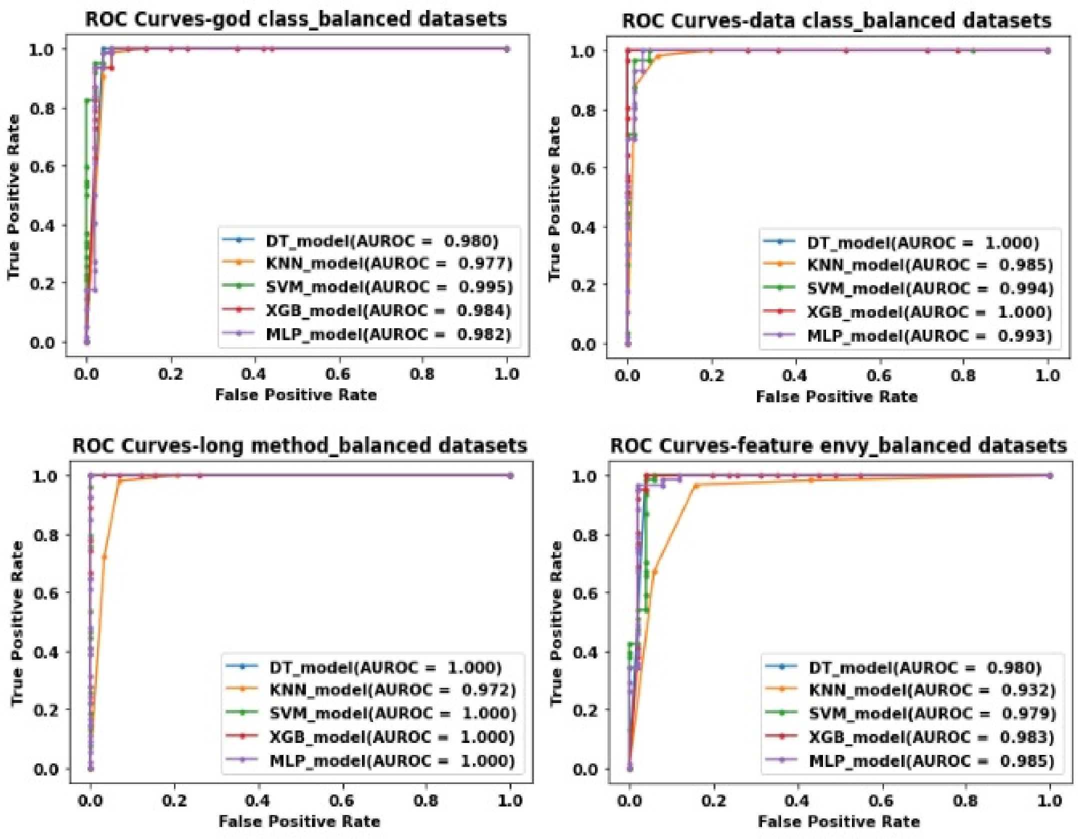

Fig. 7. The ROC curves obtained by the models on all considered code smells_ balanced datasets.

The performance of our models was compared with those of previous studies based on accuracy and AUC. The comparison results are presented in Table 10 and Table 11, where the best values are indicated in bold, and "- " denotes the missing performance measures for specific methods in certain datasets. Overall, our method outperforms the other state-of-the-art methods in most cases. A comparison of machine learning techniques using four datasets of code smells, with a specific emphasis on their Area Under the Curve (AUC) metrics. The AUC is a fundamental statistic used in binary classification. The examination focuses on the evaluation of Random Forest, Naive Bayes, SVM, and K-Nearest Neighbors (KNN) algorithms. Additionally, the study investigates Our models sing Decision Tree (DT), K-NN, Support Vector Machines (SVM), XGBoost (XGB), and Multilayer Perceptron (MLP). The AUC values are presented for the software code metrics known as "god class," "data class," "long method," and "feature envy." Our models section demonstrates excellent performance, particularly when applied to datasets that are balanced. The AUC values ranges from 0.97 to 1.00, which signifies a high level of discriminatory ability. The results presented in this study showcase the potential of Our models in detecting software development code quality issues, despite the lack of information regarding the datasets and code quality metrics utilized.

Table 10. Outlines a comparison between the proposed method and other pre-existing methods, with emphasis on their respective accuracy values

	Datasets			
Methods	*god class*	*data class*	*long method*	*feature envy*
DT [1]	-	-	-	0.97
RF [1]	-	0.99	0.95	-
RF [9]	0.96	0.98	0.99	0.96
NB [9]	0.97	0.97	0.97	0.91
NB [17]	0.96	-	0.97	0.91
MLP [17]	0.97	-	0.99	0.92
DT [17]	**0.98**	-	0.97	0.95
RF [19]	0.76	0.81	0.60	0.66
NB [19]	0.74	0.66	0.74	0.76
SVM [19]	0.66	0.66	0.66	0.60
K-NN [21]	0.97	0.97	0.97	0.91
NB [21]	0.96	0.84	0.95	0.92
MLP [21]	0.97	0.97	0.96	0.95
DT [21]	0.97	0.98	0.98	**0.98**
RF [21]	0.97	0.98	0.99	0.97
Logistic Regression [21]	0.97	0.97	0.99	0.97
RF [22]	0.69	0.70	0.68	0.71
NB [22]	0.82	0.75	0.81	0.83
SVM [22]	0.74	0.83	0.81	0.83
K-NN [22]	0.80	0.82	0.81	0.82
Our models (DT, K-NN, SVM, XGB, MLP) - Original Datasets	0.95, 0.90, 0.92, **0.98**, 0.93	0.98, 0.89, 0.96,**1.00**, 0.98	0.99, 0.92, 0.98, **1.00**, 0.94	0.92, 0.86, 0.90, 0.95, 0.88
Our models (DT, K-NN, SVM, XGB, MLP) - Balanced Datasets	**0.98**, 0.97, 0.96, 0.96, 0.97	**1.00**, 0.96, 0.97, **1.00**, 0.98	**1.00**, 0.96, **1.00**, **1.00**, 0.98	**0.98**, 0.91, 0.96, **0.98**, 0.96

Table 11. Outlines a comparison between the proposed method and other pre-existing methods, with emphasis on their respective AUC values

	Datasets			
Methods	*god class*	*data class*	*long method*	*feature envy*
RF [22]	0.59	0.65	0.52	0.59
NB [22]	0.88	0.85	0.86	0.86
SVM [22]	0.65	0.88	0.66	0.82
K-NN [22]	0.83	0.86	0.86	0.83
Our models (DT, K-NN, SVM, XGB, MLP) - Original Datasets	0.94, 0.94, 0.97, **0.99, 0.99**	0.95, 0.97, 0.99, **1.00**, 0.99	0.98, 0.94, 0.99, **1.00**, 0.98	0.90, 0.85, 0.96, **0.99**, 0.90
Our models (DT, K-NN, SVM, XGB, MLP) - Balanced Datasets	0.98, 0.97, **0.99**, 0.98, 0.98	**1.00**, 0.98, 0.99, **1.00**, 0.99	**1.00**, 0.97, **1.00**, **1.00, 1.00**	0.98, 0.93, 0.97, 0.98, 0.98

After analysing the outcomes generated by our presented ML models across all datasets, It is clear from the results that the models achieved impressive scores on all of the datasets. This suggests that our proposed models performed well, and the data balancing methods utilized were instrumental in enhancing the accuracy of ML models for code smells detection.

4. Conclusion

Code smells detection has significant positive effects on software quality. In this study, we presented a method based on ML techniques combined with a data balancing method (random oversampling technique) to detect code smells; Our proposed method was evaluated by considering four different types of code smells. The evaluation involved conducting experiments using five different ML algorithms and assessing the results using various performance measures. The proposed models' average accuracy on the original datasets was found to be 93% for god class, 96% for data class, 96% for long method, and 90% for feature envy. Meanwhile, on the balanced datasets, the proposed models' average accuracy was 96% for god class, 98% for data class, 98% for long method, and 95% for feature envy. The results indicate that the proposed models' accuracy improved by 3%, 2%, 2%, and 5% on the balanced datasets compared to the original datasets. The experimental result showed that combining ML algorithms with a random oversampling technique can enhance the process of code smells detection and that software metrics play significant and critical roles in detecting code smells. By analyzing the results, it is clear that our method gives better results when compared with the other methods previously studied for code smells detection in performance evaluations using the same datasets. However, the limitation of this study is that some of the results obtained with our models (K-NN and MLP) are not very high. So, in our future work, we will try to improve the architecture of these models to get better results. In addition, we intend to assess the robustness of our method by testing it on various datasets. Furthermore, our goal is to enhance the accuracy of models in detecting code smells by incorporating additional ML algorithms, such as neural networks and deep learning, and utilizing random under sampling techniques for data balancing.

Acknowledgment

The University of Miskolc, Faculty of Mechanical Engineering and Informatics, Institute of Information Science, provided financial support for this study, which the authors gratefully acknowledge.

Declarations

Author contribution. Both authors collaborated in developing the work, analyzing and interpreting the results, and making significant contributions to the final manuscript. The authors have also carefully reviewed and granted their approval for the publication of the final manuscript.
Funding statement. The University of Miskolc provides open Access funding.
Conflict of interest. The authors declare no conflict of interest.
Additional information. No additional information is available for this paper.

Data and Software Availability Statements

The datasets utilized in our current study can be accessed at https://essere.disco.unimib.it/machine-learning-for-code-smell-detection/.

References

[1] M. Y. Mhawish and M. Gupta, "Predicting Code Smells and Analysis of Predictions: Using Machine Learning Techniques and Software Metrics," *J. Comput. Sci. Technol.*, vol. 35, no. 6, pp. 1428–1445, Nov. 2020, doi: 10.1007/S11390-020-0323-7.

[2] 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," *MaLTeSQuE 2019 - Proc. 3rd ACM SIGSOFT Int. Work. Mach. Learn. Tech. Softw. Qual. Eval. co-located with ESEC/FSE 2019*, pp. 19–24, Aug. 2019, doi: 10.1145/3340482.3342744.

[3] N. A. A. Khleel and K. Nehéz, "Deep convolutional neural network model for bad code smells detection based on oversampling method," *Indones. J. Electr. Eng. Comput. Sci.*, vol. 26, no. 3, pp. 1725–1735, Jun. 2022, doi: 10.11591/IJEECS.V26.I3.PP1725-1735.

[4] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow, "Code Smells Detection and Visualization: A Systematic Literature Review," *Arch. Comput. Methods Eng.*, vol. 29, no. 1, pp. 47–94, Jan. 2022, doi: 10.1007/S11831-021-09566-X.

[5] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," *J. Syst. Softw.*, vol. 176, p. 110936, Jun. 2021, doi: 10.1016/J.JSS.2021.110936.

[6] A. Kaur, S. Jain, S. Goel, and G. Dhiman, "A Review on Machine-learning Based Code Smell Detection Techniques in Object-oriented Software System(s)," *Recent Adv. Electr. Electron. Eng. (Formerly Recent Patents Electr. Electron. Eng.*, vol. 14, no. 3, pp. 290–303, Sep. 2020, doi: 10.2174/2352096513999200922125839.

[7] A. Al-Shaaby, H. Aljamaan, and M. Alshayeb, "Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review," *Arab. J. Sci. Eng.*, vol. 45, no. 4, pp. 2341–2369, Apr. 2020, doi: 10.1007/S13369-019-04311-W.

[8] P. Kokol, M. Kokol, and S. Zagoranski, "Code smells: A Synthetic Narrative Review," *Libr. Philos. Pract.*, vol. 2020, pp. 1-13, Mar. 2021. [Online]. Available: https://arxiv.org/abs/2103.01088v1

[9] 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.

[10] T. Guggulothu and S. A. Moiz, "Code smell detection using multi-label classification approach," *Softw. Qual. J.*, vol. 28, no. 3, pp. 1063–1086, Sep. 2020, doi: 10.1007/S11219-020-09498-Y.

[11] J. Nanda and J. K. Chhabra, "SSHM: SMOTE-stacked hybrid model for improving severity classification of code smell," *Int. J. Inf. Technol.*, vol. 14, no. 5, pp. 2701–2707, Aug. 2022, doi: 10.1007/S41870-022-00943-8.

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

[13] F. C. Luiz, B. R. De Oliveira, and F. S. Parreiras, "Machine learning techniques for code smells detection: An empirical experiment on a highly imbalanced setup," *ACM Int. Conf. Proceeding Ser.*, May 2019, doi: 10.1145/3330204.3330275.

[14] D. Oliveira, W. K. G. Assunção, L. Souza, W. Oizumi, A. Garcia, and B. Fonseca, "Applying Machine Learning to Customized Smell Detection: A Multi-Project Study," *ACM Int. Conf. Proceeding Ser.*, pp. 233–242, Oct. 2020, doi: 10.1145/3422392.3422427.

[15] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?," *25th IEEE Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2018 - Proc.*, vol. 2018-March, pp. 612–621, Apr. 2018, doi: 10.1109/SANER.2018.8330266.

[16] D. Cruz, A. Santana, and E. Figueiredo, "Detecting bad smells with machine learning algorithms: An empirical study," *Proc. - 2020 IEEE/ACM Int. Conf. Tech. Debt, TechDebt 2020*, pp. 31–40, Jun. 2020, doi: 10.1145/3387906.3388618.

[17] A. D. F. Martins, C. Melo, J. M. Monteiro, and J. de Castro Machado, "Empirical study about class change proneness prediction using software metrics and code smells," *ICEIS 2020 - Proc. 22nd Int. Conf. Enterp. Inf. Syst.*, vol. 1, pp. 140–147, 2020, doi: 10.5220/0009410601400147.

[18] M. Hozano, N. Antunes, B. Fonseca, and E. Costa, "Evaluating the accuracy of machine learning algorithms on detecting code smells for different developers," *ICEIS 2017 - Proc. 19th Int. Conf. Enterp. Inf. Syst.*, vol. 2, pp. 474–482, 2017, doi: 10.5220/0006338804740482.

[19] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "On the Feasibility of Transfer-learning Code Smells using Deep Learning," *ACM Trans. Softw. Eng. Methodol. 1, 1, Artic.*, vol. 1, pp. 1-34, Apr. 2019, doi: 10.48550/arXiv.1904.03031.

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

[21] S. Jain and A. Saha, "Rank-based univariate feature selection methods on machine learning classifiers for code smell detection," *Evol. Intell.*, vol. 15, no. 1, pp. 609–638, Mar. 2022, doi: 10.1007/S12065-020-00536-Z.

[22] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, "A large empirical assessment of the role of data balancing in machine-learning-based code smell detection," *J. Syst. Softw.*, vol. 169, p. 110693, Nov. 2020, doi: 10.1016/J.JSS.2020.110693.

[23] E. Tempero *et al.*, "The Qualitas Corpus: A curated collection of Java code for empirical studies," *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC*, pp. 336–345, 2010, doi: 10.1109/APSEC.2010.46.

[24] G. Saranya, H. Khanna Nehemiah, A. Kannan, and V. Nithya, "Model level code smell detection using EGAPSO based on similarity measures," *Alexandria Eng. J.*, vol. 57, no. 3, pp. 1631–1642, Sep. 2018, doi: 10.1016/J.AEJ.2017.07.006.

[25] W. Xu and X. Zhang, "Multi-granularity code smell detection using deep learning method based on abstract syntax tree," *Proc. Int. Conf. Softw. Eng. Knowl. Eng. SEKE*, vol. 2021-July, pp. 503–509, 2021, doi: 10.18293/SEKE2021-014.

[26] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," *IEEE Int. Conf. Progr. Compr.*, vol. 2019-May, pp. 93–104, May 2019, doi: 10.1109/ICPC.2019.00023.

[27] M. Hadj-Kacem and N. Bouassida, "A hybrid approach to detect code smells using deep learning," *ENASE 2018 - Proc. 13th Int. Conf. Eval. Nov. Approaches to Softw. Eng.*, vol. 2018-March, pp. 137–146, 2018, doi: 10.5220/0006709801370146.

[28] N. A. A. Khleel and K. Nehez, "Comprehensive Study on Machine Learning Techniques for Software Bug Prediction," *Int. J. Adv. Comput. Sci. Appl.*, vol. 12, no. 8, pp. 726–735, 2021, doi: 10.14569/IJACSA.2021.0120884.

[29] S. Jain and A. Saha, "Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection," *Sci. Comput. Program.*, vol. 212, p. 102713, Dec. 2021, doi: 10.1016/J.SCICO.2021.102713.

[30] Z. Agusta, Z. P. Agusta, and A. Adiwijaya, "Modified balanced random forest for improving imbalanced data prediction," *Int. J. Adv. Intell. Informatics*, vol. 5, no. 1, pp. 58–65, Mar. 2019, doi: 10.26555/ijain.v5i1.255.

[31] Z. M. Zain *et al.*, "Predicting breast cancer recurrence using principal component analysis as feature extraction: an unbiased comparative analysis," *Int. J. Adv. Intell. Informatics*, vol. 6, no. 3, pp. 313–327, Nov. 2020, doi: 10.26555/ijain.v6i3.462.