Improving Accuracy of Code Smells Detection with Data Balancing Techniques

Nasraldeen Alnor Adam Khleel[1]* and Károly Nehéz[2]†
[1]*Department of Information Engineering, University of Miskolc, Miskolc, H-3515, Hungary.
[2]Department of Information Engineering, University of Miskolc, Miskolc, H-3515, Hungary.

*Corresponding author(s). E-mail(s): nasr.alnor@uni-miskolc.hu;
Contributing authors: aitnehez@uni-miskolc.hu;
†These authors contributed equally to this work.

Abstract

Code smells are indicators of potential symptoms or problems in software due to inefficient design or incomplete implementation. These problems can affect software quality in the long term. Code smell detection is fundamental to improving software quality and maintainability, reducing the risk of software failure, and helping to refactor the code. Data imbalance is the main challenge of machine learning (ML) techniques in detecting the code smells. Several prediction methods have been applied for code smells detection in our previous works. However, many of them show that, ML methods is not always suitable for code smells detection due to the problem of highly unbalanced data. To overcome these challenges, the objective of this study is to present a code smells detection method based on ML models with data balancing techniques to mitigate data unbalancing issues by taking a corpus of Java projects as experimental datasets. In our experiments, we have used Bidirectional Long Short-Term Memory (Bi-LSTM) and Gated Recurrent Unit (GRU). The performance of these models has been evaluated based on accuracy, precision, recall, f-measure, receiver operating characteristic (ROC) curve. The results indicate that the use of data balancing techniques had a positive effect on the predictive accuracy of the models presented.

Keywords: code smells, software metrics, deep learning (DL), Bi-LSTM, GRU, class imbalance, over-sampling techniques.

Introduction

Software development and maintenance involve high costs, especially as systems become larger and more complex. Code smells indicate design or programming issues, which make it difficult to alter and maintain the software [1, 2, 3]. Code smells are one of the most accepted approaches to identify design problems in the source code and detection of code smells is a very important step for guiding the subsequent steps in the refactoring process. The quality of the software can be enhanced if code smells are identified in the class and method levels in the source code. Several studies examined the impact of code smells on software [4, 5, 6], and they showed their adverse effects on the quality of the software. Researchers have developed many techniques and conducted many experimental studies to detect code smells and obtained different results when applying the same case study [6]. Class imbalance is one of the most common problems for classification models during training and validation. There are many data sampling techniques that are used to deal with imbalanced class distributions such as oversampling and under-sampling techniques. The oversampling techniques supplements instances of the minority class to the dataset, while the under-sampling techniques eliminate samples of the majority class for the goal of obtaining a balanced dataset. Random Oversampling and Tomek Links techniques are the most widely used sampling techniques to solve the class imbalance problem. Random over-sampling is a technique developed to increase the size of a training data set by making multiple copies of some minority classes, Tomek Links is an under-sampling technique developed to randomly select samples with its k-nearest neighbours from the majority class that wants to be removed [7, 8]. DL is a special type of machine learning technique that allows computational models consisting of multiple processing layers to learn to represent data with multiple levels of abstraction. DL architecture has been widely used to solve many detections, classification, and prediction problems [9, 10, 11]. Software metrics are essential aids to measure and improve software quality, and these metrics are used to measure and characterize software engineering products. Metrics are considered to be independent variables, which means that they are used to perform the prediction (i.e., the predictors) [12, 13]. The main role of software metrics is to estimate and measure some characteristics of systems such as size, complexity, inheritance, encapsulation, etc. [14, 15]. Selected metrics are a large set of object-oriented metrics that are considered as independent variables as shown in Table 1. This paper aims to investigate ML/DL and sampling techniques to build a predictive model that can detect different types of code smells in source codes. Four different code smells were used to evaluate the capability of the proposed model based on various performance measures. This paper is organized as follows: first part contains a discussion on related work, and we present the background on the topics of code smells. After that, our research methodology is presented. The experimental results and discussion are followed by the conclusions.

Table 1. Selected metrics in this study [4].

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

All metrics have been calculated with software tools, which analyse the source code of Java projects using the Eclipse JDT library. These tools are Design Features and Metrics for Java (DFMC4J), which have been designed to be integrated as a library into other projects.

Related work

There are many previous studies in the field of code smells detection based on different methodologies and strategies [12, 16, 31]. Hui Liu et al. [9] proposed a new DL-based approach to detecting code smells. The approach is evaluated based on four types of code smell: feature envy, long method, large class, and misplaced class. The experiment results show that the proposed approach significantly improves the state-of-the-art. Tushar Sharma et al. [10] proposed Convolution Neural Networks (CNN) and Recurrent Neural Network (RNN) models for code smell detection. They performed the training and evaluation on C# sample codes. The experiment results show that, it is feasible to detect smells using DL methods and transfer-learning is possible to detect code smells with a performance similar to that of direct learning. Tushar Sharma et al. [12] proposed DL methods based on CNN and RNN for code smell detection. Models were evaluated based on C# and Java code samples and the results show that it is possible to detect smells using DL methods. Dong Kwan Kim [15] proposed a neural network model-based code smells detection system using a set of Java projects as an experimental dataset. The empirical results showed that the prediction outcomes are improved more when the model is highly trained with more datasets. Hadj-Kacem and Nadia. [16] proposed a hybrid approach based on Deep Autoencoder and artificial neural network algorithms to detect code smells. The approach was evaluated based on four code smells extracted from 74 open-source systems. The experiment results show that the values of recall and precision measurements have demonstrated high accuracy results. Jaspreet Kaur and Satwinder Singh [17] suggested a neural network model based on object-oriented metrics for the detection of bad code smells. The model was trained and tested using different epochs to find twelve bad code smells. The experimental results showed the relationship between bad smells and object-oriented metrics. Hui Liu et al. [18] proposed a DL-based novel approach to detecting feature envy. The approach has been evaluated on open-source applications. The results show that the proposed approach significantly improves the state-of-the-art. Weiwei Xu and Xiaofang Zhang [19] proposed a novel DL approach based on abstract syntax trees (ASTs) to detect multi-granularity code smells based on four types of smells. Experimental results show that this approach yields better results than the latest methods for detecting code smells with different granularities. Ananta Kumar Das et al. [20] proposed a DL-based approach to detect two code smells (Brain Class and Brain Method). The proposed system is evaluated based on thirty open-source Java projects from GitHub repositories. The experiments demonstrated high accuracy results for both the code smells.

Background

In this section, we present the background of the topics of code smells, Bidirectional LSTM, and Gated Recurrent units.

Code smells

Code smells are symptoms of code and design problems (Fowler and beck) [21]. Where code smells refer to any anomaly in the source code that shows a violation of basic design rules such as abstraction, hierarchy encapsulation, etc. Code smells represent design choices that may lead to a future degradation of maintainability, understanding ability, and changeability of a given part of the code [22, 23]. So, code smells can lead to shortcomings in software that make software hard to evolve and maintain and trigger refactoring of code [24]. As dependent variables choose code smells with high frequency that may have the greatest negative impact on the software quality, which can be recognized by some available detection tools [25, 26]. Code smells detection is the primary requirement to guide the subsequent steps in the refactoring process [16]. Detection rules are approaches used to detect code smells through a combination of different software metrics with predefined threshold values. Most of the current detectors need the specification of thresholds that allow them to distinguish smelly and non-smelly code [27]. Many approaches have been presented by the authors for uncovering the smells from the software systems. There is a different type of detection methodologies that 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 certain 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 DL to detect code smells to avoid thresholds and decrease the false positive rate in code smells detection tools [28, 29]. The following four code smells and detector tools were considered in this study as shown in Table 2.

Table 2. Code smells and detector tools are considered in the study based on class and method project level.

Code smells	**Description**	**Affected entity**	**Detectors**
God Class	Classes have many members and implement different behaviors.	Class	iPlasma, PMD

Data Class	Classes contain only fields (data) and methods for accessing them.	Class	iPlasma, Fluid Tool, Antipattern Scanner
Feature envy	Sign of breach of the rule of grouping behavior with related data happens when a method is more interested in other properties of the classes than in the ones from its class.	Method	iPlasma, PMD, Marinescu.
Long method	Refers to the too-long method. It is classified as a blotter smell that affects method-level entities.	Method	iPlasma, Fluid Tool [4].

Table 2 shows the four code smells and detector tools are considered to identify the presence of code smells in the given. These smells are closely related to the class level smells and method level smells.

Bidirectional LSTM

The idea behind Bidirectional Recurrent Neural Networks (BI-RNNs) is to exploit spatial features to capture bidirectional temporal dependencies from historical data to overcome the limitations of traditional RNNs. All input information available in the past and future is used to train the BI-RNNs model as shown in Figure 1. The first step in BI-RNNs is to identify of information by a sigmoid layer called the forget gate layer that controls how much information flows from the inputs to the state and from the state to the outputs based on current inputs and previous outputs. Standard RNNs take sequences as inputs, and each step of the sequence refers to a certain moment [10, 12, 22]. For a certain moment t, the output o_t not only depends on the current input x_t but is also influenced by the output from the previous moment $t-1$. The following equations show the output of moment (t).

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

Where U, V, and W denote the weights of the RNN, b and c denote the bias, f and g are the activation functions of the neurons. The cell state carries the information from the previous moments and will flow through the entire LSTM chain, which is the key that LSTM can have long-should be filtered from the previous moment, the output of forget gate can be formulated as the following equation:

$$f_t = \sigma(W_f \,.\, [h_{t-1\,,}\, x_t] + b_f) \tag{2}$$

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

$$i_t = \sigma(W_i \,.\, [h_{t-1\,,}\, x_t] + b_i) \tag{3}$$

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

$$\check{C}_t = \tanh(W_c \,.\, [h_{t-1\,,}\, x_t] + b_c) \tag{4}$$
$$\check{C}_t = f_t \times C_{t-1} + i \times \check{C}_t)$$

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

$$o_t = \sigma(W_o \,.\, [h_{t-1\,,}\, x_t + b_o] \tag{5}$$
$$h_t = o_t \times \tanh(C_t).$$

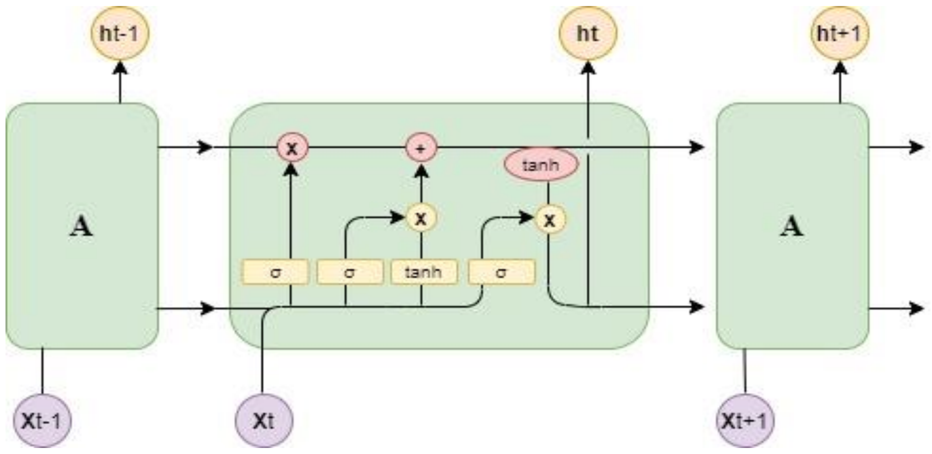

Figure 1. Interacting layers of the repeating module in a Bi-LSTM.

Gated Recurrent Unit

With the increasing applications of Recurrent Neural Networks (RNN), it is found that there is a long-term dependencies problem. There are gradient vanishing and gradient explosion phenomena when learning sequence. This problem means RNN cannot guarantee the long-term nonlinear relationship. For this reason, many optimization theories have been introduced and many optimization algorithms have been derived such as long short-term memory networks (LSTM), GRU networks, Bi-LSTM, Echo State Networks, and independent RNN [12, 22]. GRU network is one of the optimized structures of the RNN. The goal of the GRU network is to solve the long-term dependence and gradient disappearance problem of RNN. A GRU is like LSTM with a forget gate, but it has fewer parameters than LSTM and uses an update gate and a reset gate as shown in figure 2. The update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future and the reset gate helps the model to decide how much of the past information to forget [10]. The update gate model in the GRU neural network is calculated as shown in the equation below.

$$z(t) = \sigma(W(z).[h(t-1), x(t)]) \quad (6)$$

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

$$r(t) = \sigma(W(r).[h(t-1), x(t)]) \quad (7)$$

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

$$\check{h}(t) = \tanh\left(W\check{h}.[rt * h(t-1), x(t)]\right) \quad (8)$$

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

$$h(t) = \left(1 - z(t)\right) * h(t-1) + z(t) * \check{h}(t) \quad (9)$$

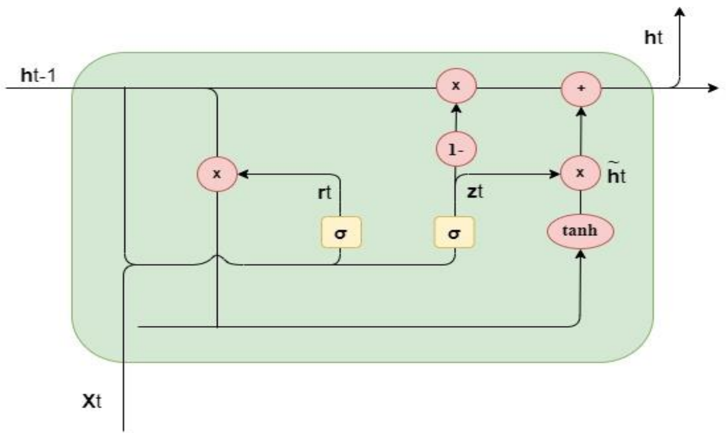

figure 2. Interacting layers of the repeating module in GRU.

Proposed approach

The main objective of this study is to perform an empirical study on the role of data balancing techniques in improving DL accuracy for the detection of code smells. The experiment used two DL models and 74 open-source systems from Qualitas Corpus (QC) of systems. The empirical study follows the methodology shown in figure 3. This study aims to address the following research questions:
RQ1: Do data balancing techniques improve ML techniques accuracy for detection of code smells?
RQ2: Which data balancing technique is the most effective at improving the accuracy of ML techniques?

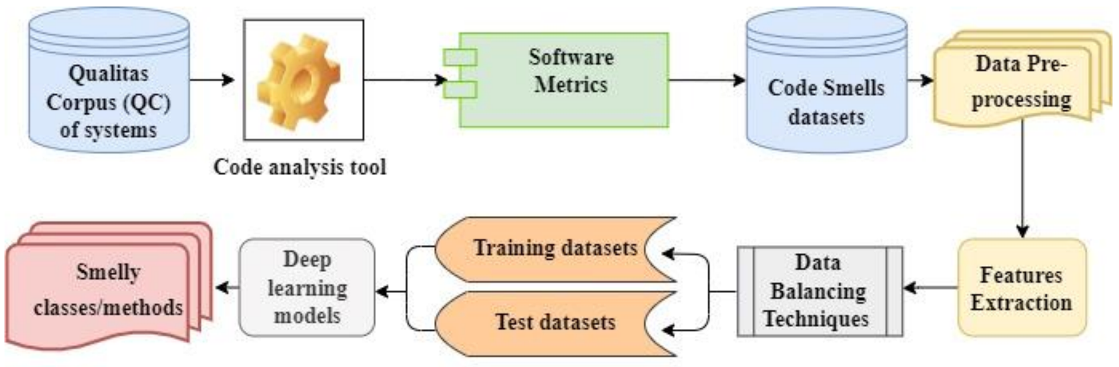

Figure 3. Overview of the Proposed Model for code smells Detection

Data modelling and collection

Dataset selection is an important task in the problem of DL, and classification models perform better if the dataset is more relevant to the problem. The code smells detection model in this study uses a supervised learning task that relies on a large set of software metrics as independent variables. Having many systems or datasets is fundamental to training DL models and allowing generalization of the obtained results. To perform the analysis and experiment, the model used the proposed datasets in Arcelli Fontana et al [4]. The authors selected 74 open-source systems from Qualitas Corpus as shown in Table 3. The Qualitas Corpus (QC) of systems collected by Tempero et al [30]. The corpus used is composed of 111 systems written in Java, characterized by different sizes and belonging to different application domains. The reason for the selection is that the systems must be compliable to correctly compute the metrics values [4].

Table 3. Summary of projects characteristics [4].

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

Data Pre-processing and Feature Selection

Pre-processing the collected data is one of the important stages before constructing the model. Not all data collected is directly used in the analysis process (train and build the model). Whatever input fits into the model will greatly impact the performance of the DL models and later further affect the output. Data pre-processing is a group of techniques that are applied to the data to improve the quality of it before model building to remove the unwanted noise and outliers from the dataset, handle missing values, convert the type of feature. Feature Selection (FS) is one of the significant pre-processing steps and plays a key role in classification tasks. FS is the process of identifying and removing irrelevant and redundant features to improve the performance of the classifier [6, 11, 32].

Class Imbalance

Class imbalance in classification models represents those situations, where the number of examples of one class are much smaller than another classes. Class imbalance problem makes classification models not effectively predict minority modules. Many techniques have been used to solve the problem of unbalanced data such as sampling techniques, bagging and boosting-based ensemble methods, and cost-sensitive learning techniques [7, 8]. In this study, the dataset chosen for the task of code smells detection is highly imbalanced. Each of the four datasets is composed of 420 instances (classes or methods), the two first datasets concern the code smells at the class level where the number of instances is 140 for God Class and Data Class. While at the method level, the number of instances for Feature Envy is 140, and the number of instances for Long Method is 140. To solve the problem of class imbalance, in this study we modified the original datasets, by modifying the distribution with the algorithms of Random Over-sampling and Tomek Links. Figure 4 shows the distribution of learning instances over original and balanced datasets.

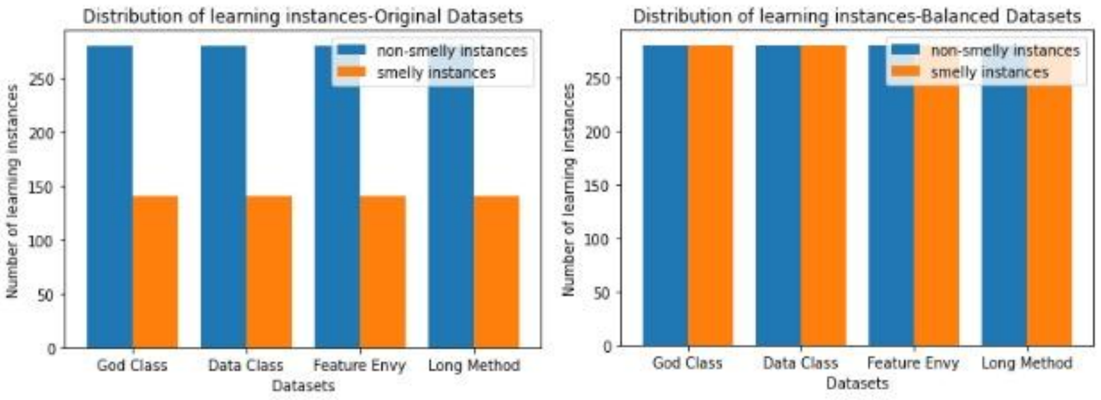

Figure 4. Distribution of learning instances over original and balanced datasets

Models building and evaluation

Different DL algorithms are used to build code smells prediction models and each algorithm has its benefits [12]. An empirical study has been conducted to demonstrate the effectiveness and accuracy of the proposed DL models for code smells detection. To get more accurate results, the proposed model has been trained and tested using a set of massive open-source Java projects. This study used a set of common performance measures based on the confusion matrixes, ROC Curve and Loos Function. The ROC curve is a graph that explains and shows the performance of classification models at all classification thresholds and plots based on two parameters, true positive rate (TPR) and false-positive rate (FPR). Loss functions are metrics used to compute the quantity of loss that a model should seek to minimize during training and quantify how good or bad the model is performing. The correlation matrix was used to describe the performance of a classification method using a set of test data. Each row of the matrix corresponds to a predicted class, whereas each column of the matrix corresponds to an actual class. The correlation matrix is a specific table that is used to measure the performance of the model. The correlation summarizes the results of the testing algorithm and presents a report of (1) True Positive Rate (TPR), (2) False Positive

Rate (FPR), (3) True Negative Rate (TNR), and (4) False Negative Rate (FNR). Table 4 shows the correlation matrix.

Table 4. Correlation matrix

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

Accuracy = (TP+TN) / (TP+FP+FN+TN) (10)

Precision = TP / (TP+FP) (11)

Recall = TP / (TP + FN) (12)

F-Measure = (2 * Recall * Precision) / (Recall + Precision) (13)

Experimental Results and Discussion

In this section, we evaluate the effectiveness and efficiency of our proposed DL models. We experimented with our method by selecting two suitable DL algorithms and testing them on the generated datasets based on four code smells. The experimental environment was based on Python environment.

Table 5. Evaluation results for the original datasets

Bi-LSTM Model						
Datasets	**Performance Measures**					
	Accuracy	**Precision**	**Recall**	**F- measure**	**ROC area**	**Loos**
God Class	0.98	0.97	0.97	0.97	0.99	0.022
Data Class	0.96	0.95	0.91	0.93	0.97	0.032
Feature envy	0.93	0.84	0.96	0.90	0.96	0.067
Long method	0.98	0.96	0.96	0.96	0.99	0.023
GRU Model						
Datasets	**Performance Measures**					
	Accuracy	**Precision**	**Recall**	**F- measure**	**ROC area**	**Loos**
God Class	0.94	1.00	0.86	0.93	0.97	0.059
Data Class	0.98	0.92	1.00	0.96	0.99	0.024
Feature envy	0.92	0.83	0.93	0.88	0.94	0.074
Long method	0.96	0.93	0.96	0.95	0.98	0.033

Table 6. Evaluation results for the balanced datasets - Random Oversampling

Bi-LSTM Model						
Datasets	**Performance Measures**					
	Accuracy	**Precision**	**Recall**	**F- measure**	**ROC area**	**Loos**
God Class	0.96	0.95	0.98	0.97	0.97	0.032
Data Class	1.00	1.00	1.00	1.00	1.00	0.001
Feature envy	0.98	0.97	1.00	0.98	0.98	0.018
Long method	1.00	1.00	1.00	1.00	1.00	0.001
GRU Model						
Datasets	**Performance Measures**					
	Accuracy	**Precision**	**Recall**	**F- measure**	**ROC area**	**Loos**
God Class	0.96	0.97	0.97	0.97	0.98	0.036

Data Class	0.99	0.98	1.00	0.99	1.00	0.015
Feature envy	0.97	0.98	0.97	0.98	0.96	0.034
Long method	0.99	0.98	1.00	0.99	0.99	0.010

Table 7. Evaluation results for the balanced datasets - Tomek links

Bi-LSTM Model						
Datasets	**Performance Measures**					
	Accuracy	**Precision**	**Recall**	**F- measure**	**ROC area**	**Loos**
God Class	0.96	1.00	0.87	0.93	0.99	0.030
Data Class	0.96	0.92	0.96	0.94	0.98	0.041
Feature envy	0.98	0.97	0.97	0.97	0.98	0.026
Long method	0.98	0.94	1.00	0.97	0.98	0.024
GRU Model						
Datasets	**Performance Measures**					
	Accuracy	**Precision**	**Recall**	**F- measure**	**ROC area**	**Loos**
God Class	0.96	0.95	0.91	0.93	0.98	0.039
Data Class	0.99	0.96	1.00	0.98	0.99	0.019
Feature envy	0.96	0.96	0.93	0.95	0.98	0.041
Long method	0.98	0.94	1.00	0.97	0.98	0.022

Results of RQ1

RQ1 Do data balancing techniques improve ML techniques accuracy for detection of code smells?
Table 5 presents the results of our Bi-LSTM and GRU Models on the original datasets in terms of accuracy, precision, recall, F-Measure, ROC area, and Loos. We notice that the accuracy values of the Bi-LSTM model vary from 0.93 to 0.98, the precision values range from 0.84 to 0.97, the recall values range from 0.91 to 0.97, the F-Measure values vary from 0.90 to 0.97, the ROC area values range from 0.96 to 0.99, and the Loos values range from 0.022 to 0.067 across all datasets. The accuracy values of the GRU model vary from 0.92 to 0.98, the precision values range from 0.83 to 1.00, the recall values range from 0.86 to 1.00, the F-Measure values vary from 0.88 to 0.96, the ROC area values range from 0.94 to 0.99, and the Loos values range from 0.024 to 0.074 across all datasets.
Table 6 presents the results of our Bi-LSTM and GRU Models on the balanced datasets using random oversampling in terms of accuracy, precision, recall, F-Measure, ROC area, and Loos. We notice that the accuracy values of the Bi-LSTM model vary from 0.96 to 1.00, the precision values range from 0.95 to 1.00, the recall values range from 0.98 to 1.00, the F-Measure values vary from 0.97 to 1.00, the ROC area values range from 0.97 to 1.00, and the Loos values range from 0.001 to 0.032 across all datasets. The accuracy values of the GRU model vary from 0.96 to 0.99, the precision values range from 0.97 to 0.98, the recall values range from 0.97 to 1.00, the F-Measure values vary from 0.97 to 0.99, the ROC area values range from 0.96 to 1.00, and the Loos values range from 0.010 to 0.036 across all datasets.
Table 7 presents the results of our Bi-LSTM and GRU Models on the balanced datasets using Tomek links in terms of accuracy, precision, recall, F-Measure, ROC area, and Loos. We notice that the accuracy values of the Bi-LSTM model vary from 0.96 to 0.98, the precision values range from 0.92 to 1.00, the recall values range from 0.87 to 1.00, the F-Measure values vary from 0.93 to 0.97, the ROC area values range from 0.98 to 0.99, and the Loos values range from 0.024 to 0.041 across all datasets. The accuracy values of the GRU model vary from 0.96 to 0.99, the precision values range from 0.94 to 0.96, the recall values range from 0.91 to 1.00, the F-Measure values vary from 0.93 to 0.98, the ROC area values range from 0.98 to 0.99, and the Loos values range from 0.019 to 0.041 across all datasets.

For each code smell, the best models obtain the following performance on the original datasets: Bi-LSTM model scores 0.98 % of accuracy on God Class and Long method. GRU model scores 0.99 % of accuracy on Data Class.
For each code smell, the best models obtain the following performance on the balanced datasets (using Random Oversampling Technique): Bi-LSTM model scores 1.00 % of accuracy on Data Class and Long method. GRU model scores 0.99 % of accuracy on the Data Class and Long method.
For each code smell, the best models obtain the following performance on the balanced datasets (using Tomek links Technique): Bi-LSTM model scores 0.98 % of accuracy on Feature envy and Long method. GRU model scores 0.99 % of accuracy on Data Class.
To answer research question **RQ1**, we compared the results obtained by the proposed models on the original datasets with results obtained by the proposed models on the balanced datasets, we note that both models got good scores on all original and balanced datasets but the results obtained by the proposed models on the balanced datasets are better, which indicated that the proposed models performed well and data balancing techniques play an important role in improving the accuracy of DL for detection of code smells.

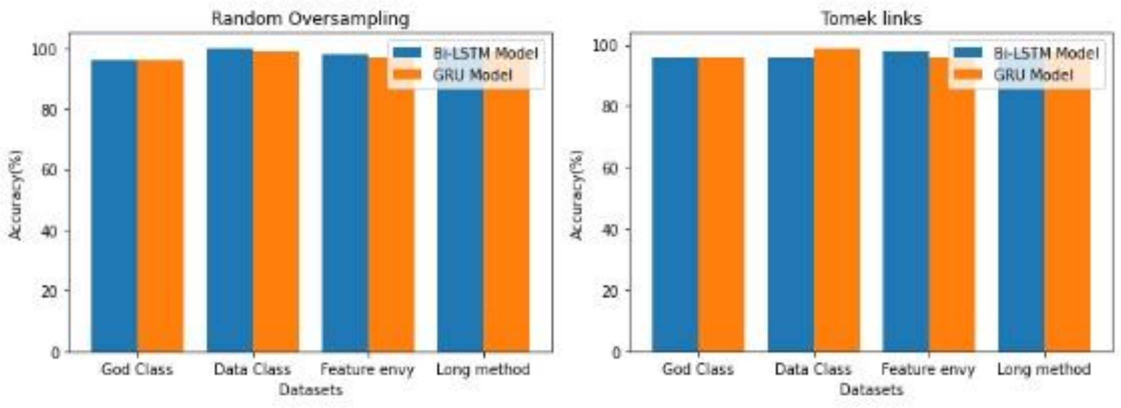

Figure 5. Accuracy of Models on Balanced Datasets

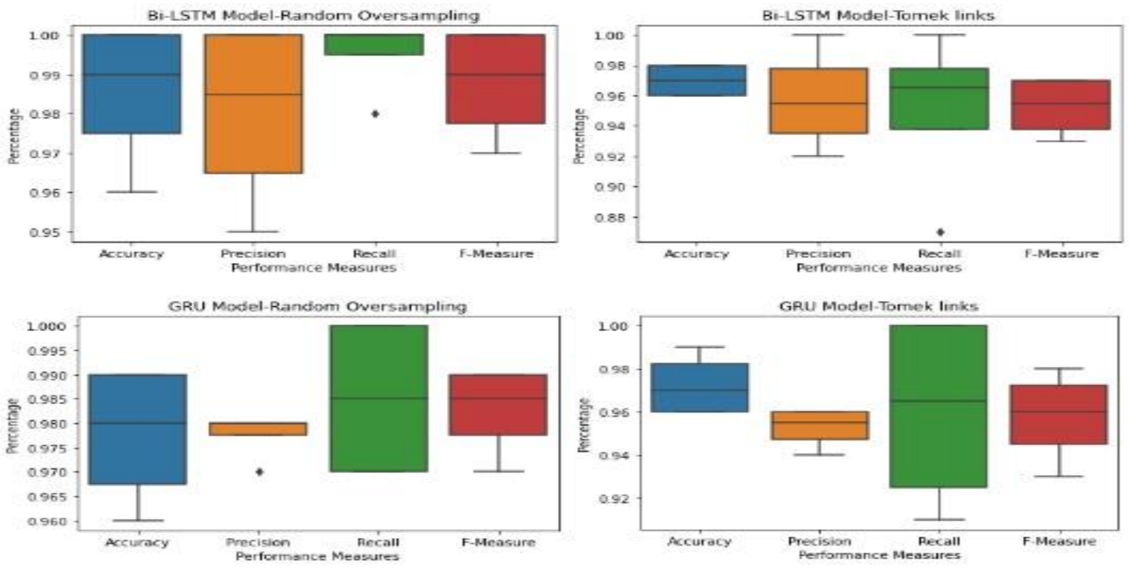

Figure 6. Boxplots representing performance measures obtained by models on balanced datasets

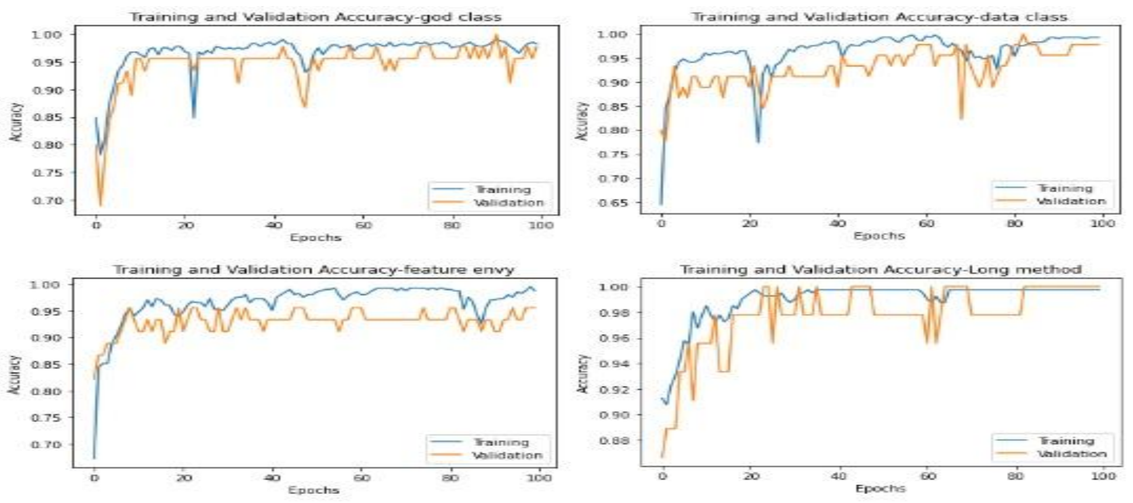

Figure 7. Training and Validation Accuracy on Balanced Datasets using Bi-LSTM Model-Random Oversampling

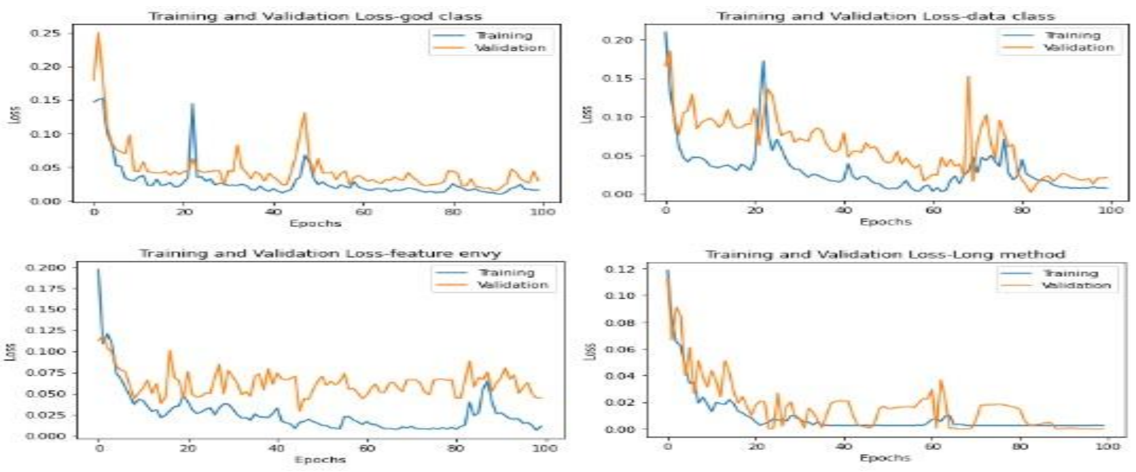

Figure 8. Training and Validation Loss on Balanced Datasets using Bi-LSTM Model-Random Oversampling

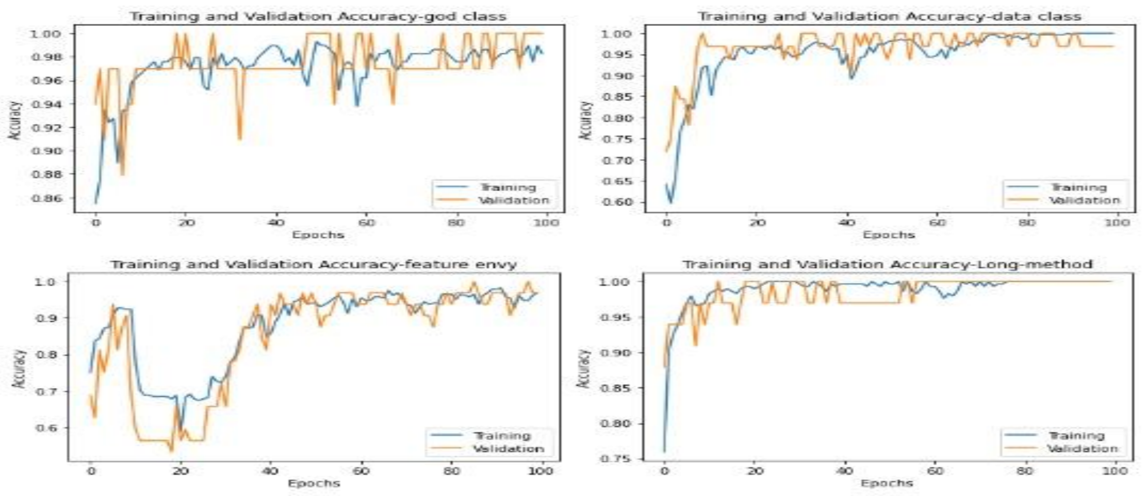

Figure 9. Training and Validation Accuracy on Balanced Datasets using Bi-LSTM Model- Tomek links

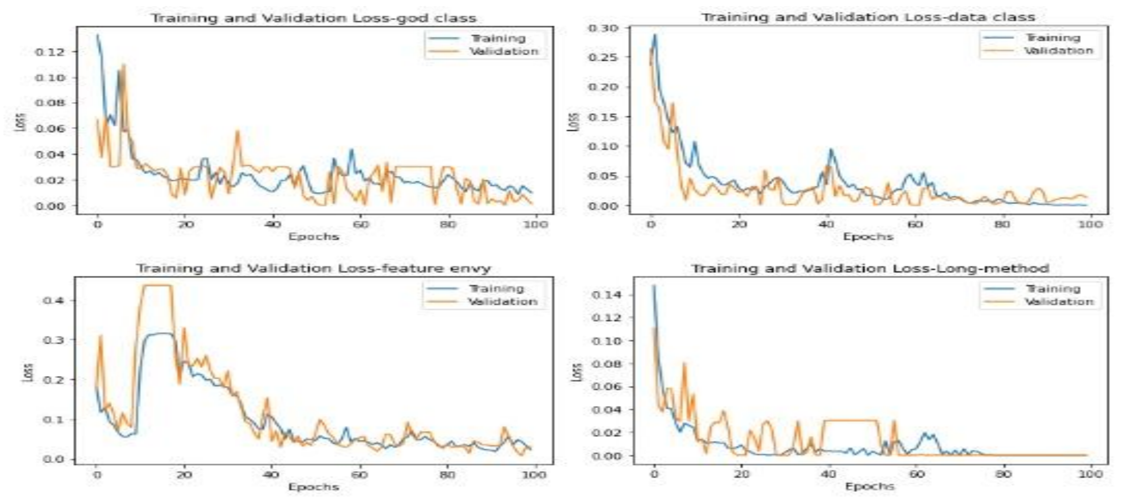

Figure 10. Training and Validation Loss on Balanced Datasets using Bi-LSTM Model- Tomek links

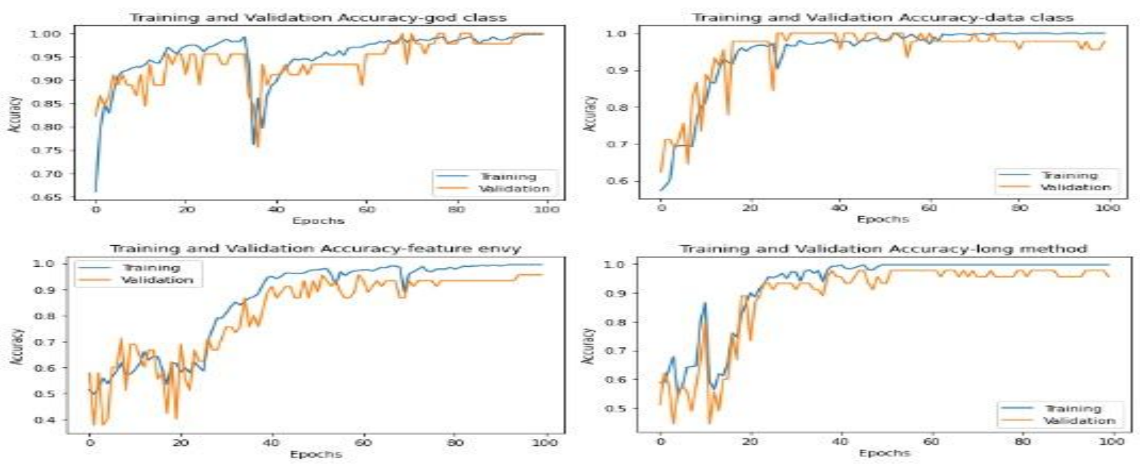

Figure 11. Training and Validation Accuracy on Balanced Datasets using GRU Model-Random Oversampling

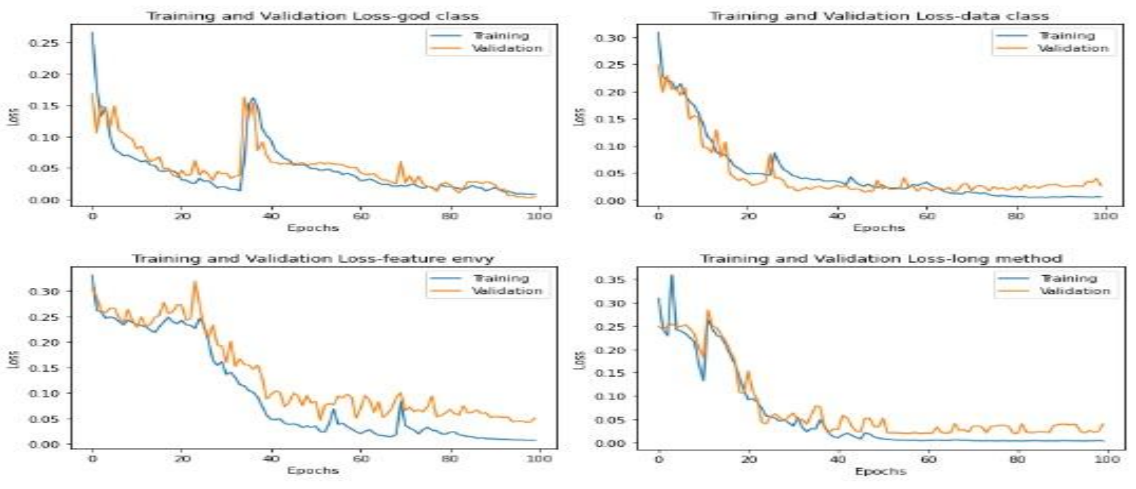

Figure 12. Training and Validation Loss on Balanced Datasets using GRU Model-Random Oversampling

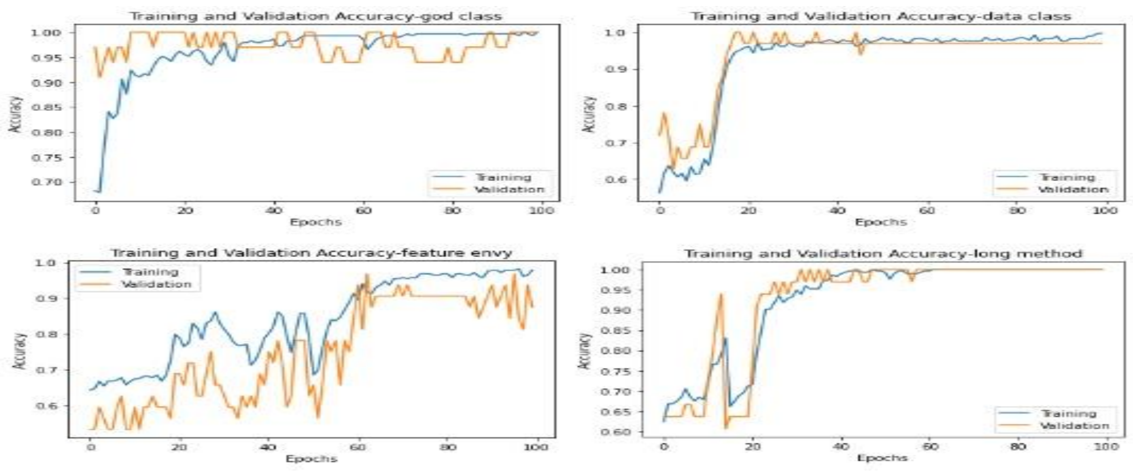

Figure 13. Training and Validation Accuracy on Balanced Datasets using GRU Model- Tomek links

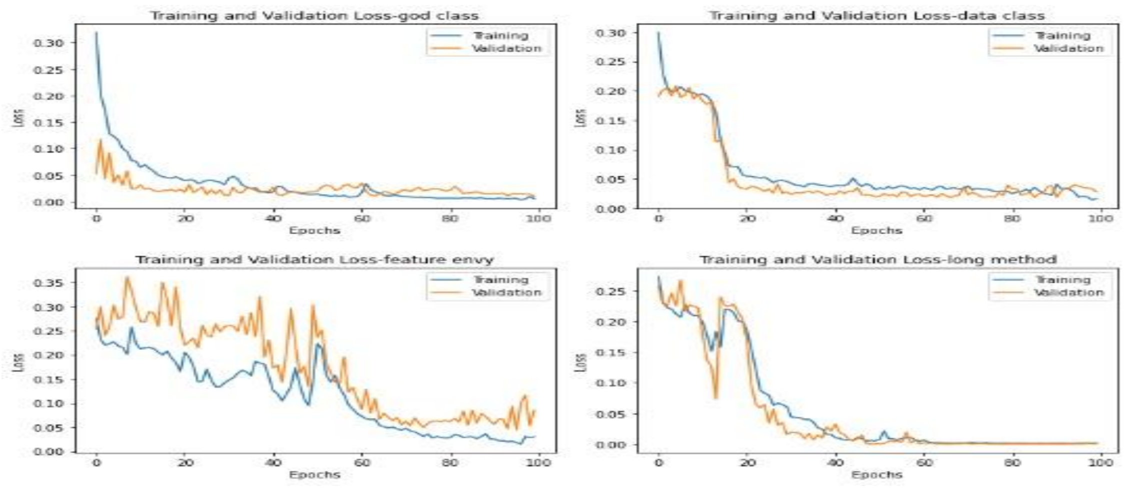

Figure 14. Training and Validation Loss on Balanced Datasets using GRU Model - Tomek links

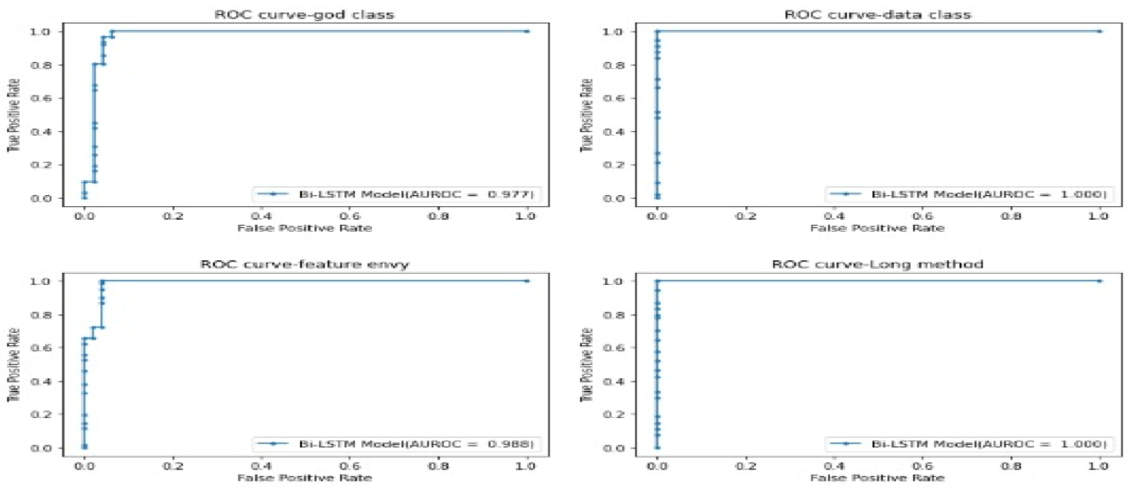

Figure 15. ROC curves for balanced datasets - Bi-LSTM Model-Random Oversampling

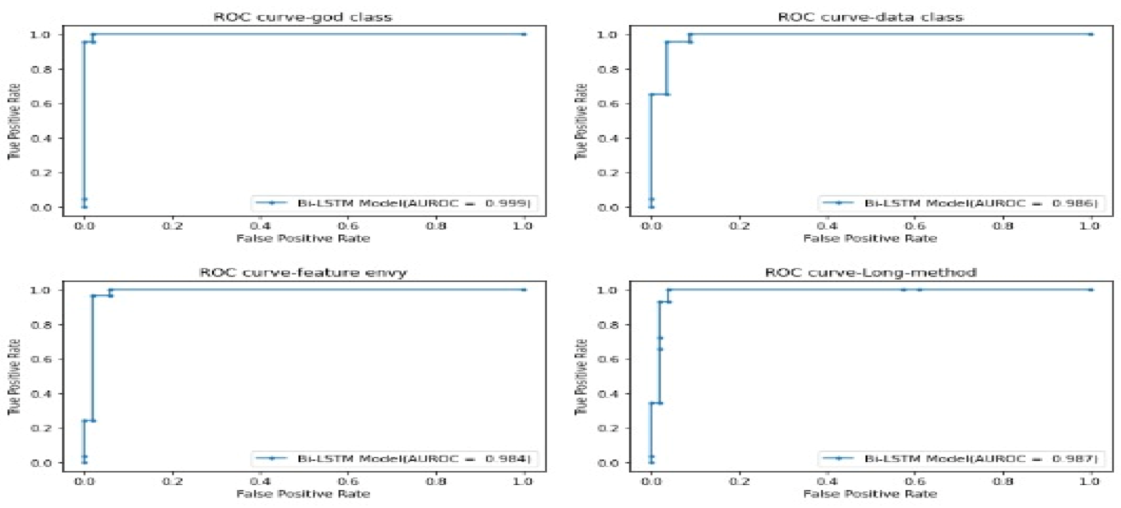

Figure 16. ROC curves for balanced datasets - Bi-LSTM Model- Tomek links

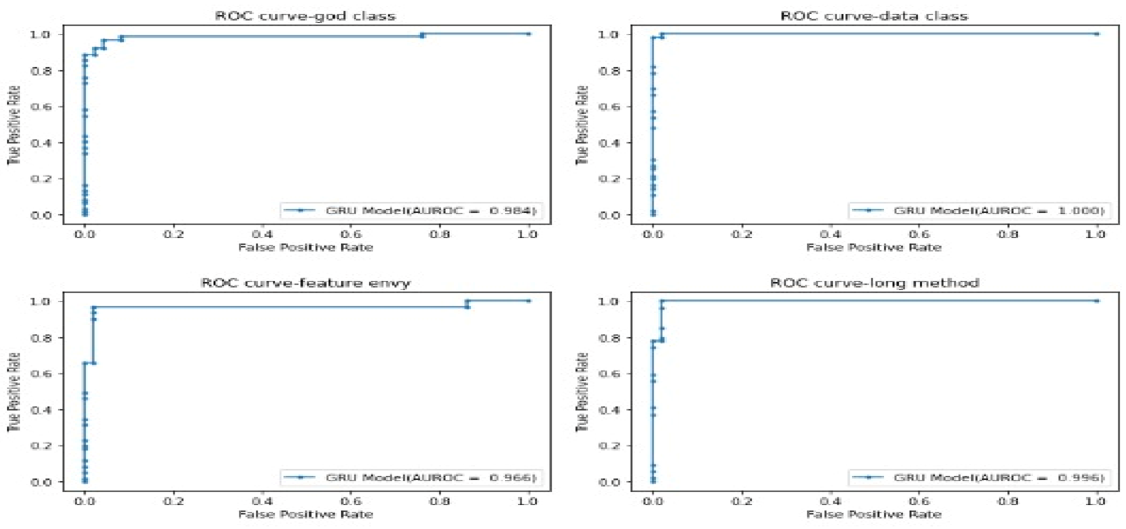

Figure 17. ROC curves for balanced datasets - GRU Model-Random Oversampling

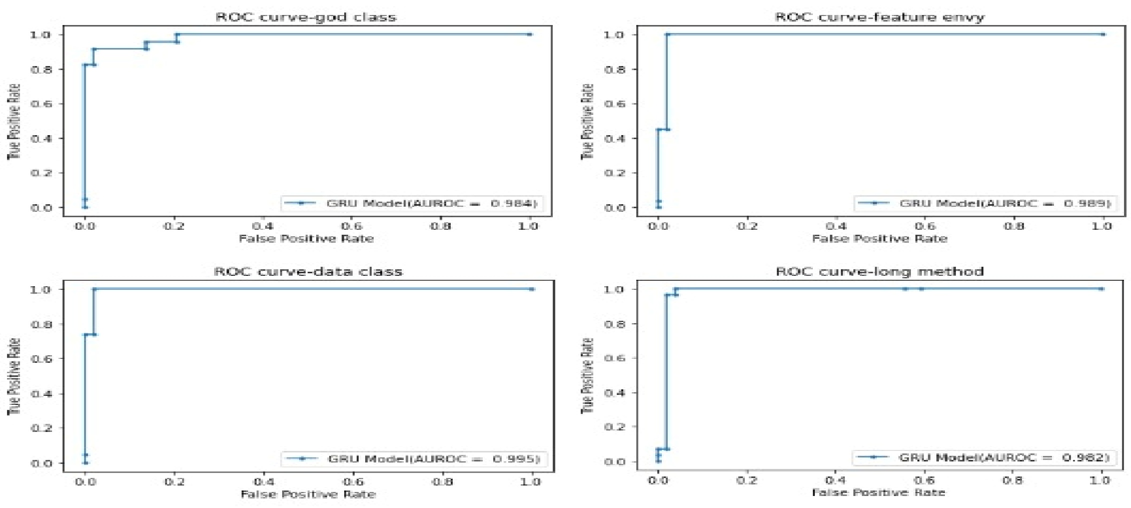

Figure 18. ROC curves for balanced datasets - GRU Model- Tomek links

Results of RQ2

RQ2 Which data balancing technique is the most effective at improving the accuracy of ML techniques?

Figure 5 shows the accuracy of models on all balanced datasets. Figure 6 shows the Box plots for the models comparing the performance measures (accuracy, precision, recall, and f-measure) of both the Bi-LSTM and GRU. Figures 7 to 14 below show the training and validation accuracy and training and validation loss of the models on all balanced datasets. Figures 15 to 18 below show the ROC curves of the models on all balanced datasets.

To answer research question **RQ2**, we compared the results obtained by the proposed models on the balanced datasets using two data balancing techniques namely Random Oversampling and Tomek links, we generally note that the Random Oversampling technique is better than the Tomek links technique.

Conclusion

Code smells in the software systems are indicators of problems that can negatively affect software quality and make it hard to maintain, reuse, and expand. This study presented a methodology based on DL and software metrics to detect code smells from software projects, considering four different types of code smells over a dataset of 6,785,568 lines of code comprising 74 open-source software systems. We conducted a set of experiments using two DL models on Java projects in different application domains and evaluated smells detection accuracy using different performance measures. The experimental result showed that the balancing techniques can improve the performance of DL techniques for code smells detection.

Abbreviations

ML: Machine Learning; Bi-LSTM: Bidirectional Long Short-Term Memory; GRU: Gated Recurrent Unit; ROC: Receiver Operating Characteristic; DL: Deep Learning; CNN: Convolution Neural Networks; RNN: Recurrent Neural Network; DFMC4J: Design Features and Metrics for Java; BI-RNNs: Bidirectional Recurrent Neural Networks; QC: Qualitas Corpus; FS: Feature Selection.

Acknowledgments

The authors gratefully acknowledge the financial assistance from the Institute of Information Science, Faculty of Mechanical Engineering and Informatics, University of Miskolc.

Author Contributions

Both authors developed the whole work, discussed the results, and contributed to the final manuscript. The authors read and approved the final manuscript.

Funding

Open Access funding provided by University of Miskolc.

Availability of data and material

The datasets for the current study are available from https://essere.disco.unimib.it/machine-learning-for-code-smell-detection/

Declarations

Competing interests

The authors declared that they have no competing of interests to this work. We declare that we do not have any commercial or associative interest that represents a conflict of interest in connection with the work submitted.

Ethics Approval and Consent to Participate

Not applicable.

Consent for Publication

Not applicable.

References

1. Kaur, Amandeep, et al. "A review on machine-learning based code smell detection techniques in an object-oriented software system (s)." Recent Advances in Electrical & Electronic Engineering (Formerly Recent Patents on Electrical & Electronic Engineering) 14.3 (2021): 290-303.
2. Lewowski, Tomasz, and Lech Madeyski. "Code Smells Detection Using Artificial Intelligence Techniques: A Business-Driven Systematic Review." Developments in Information & Knowledge Management for Business Applications (2022): 285-319.
3. Virmajoki, Joonas. "Detecting code smells using artificial intelligence: a prototype." (2020).
4. Fontana, Francesca Arcelli, et al. "Comparing and experimenting machine learning techniques for code smell detection." Empirical Software Engineering 21.3 (2016): 1143-1191.
5. Guggulothu, Thirupathi, and Salman Abdul Moiz. "Code smell detection using multi-label classification approach." Software Quality Journal 28.3 (2020): 1063-1086.
6. Mhawish, Mohammad Y., and Manjari Gupta. "Predicting Code Smells and Analysis of Predictions: Using Machine Learning Techniques and Software Metrics." Journal of Computer Science and Technology 35.6 (2020): 1428-1445.
7. Pecorelli, Fabiano, et al. "On the role of data balancing for machine learning-based code smell detection." Proceedings of the 3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation. 2019.
8. Pecorelli, Fabiano, et al. "A large empirical assessment of the role of data balancing in machine-learning-based code smell detection." Journal of Systems and Software 169 (2020): 110693.
9. Liu, Hui, et al. "Deep learning-based code smell detection." IEEE transactions on Software Engineering (2019).
10. Sharma, Tushar, et al. "On the feasibility of transfer-learning code smells using deep learning." arXiv preprint arXiv:1904.03031 (2019).
11. Fakhoury, Sarah, et al. "Keep it simple: Is deep learning good for linguistic smell detection?." 2018 IEEE 25th International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, 2018.
12. Sharma, Tushar, et al. "Code smell detection by deep direct-learning and transfer-learning." Journal of Systems and Software 176 (2021): 110936.
13. Padhy, Neelamadhab, Suresh Satapathy, and R. Singh. "Utility of an object-oriented reusability metrics and estimation complexity." Indian J. Sci. Technol 10.3 (2017): 1-9.
14. Alkharabsheh, Khalid, et al. "A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of God class." Information and Software Technology 143 (2022): 106736.
15. Kim, Dong Kwan. "Finding bad code smells with neural network models." International Journal of Electrical and Computer Engineering 7.6 (2017): 3613.

16. Hadj-Kacem, Mouna, and Nadia Bouassida. "A Hybrid Approach To Detect Code Smells using Deep Learning." ENASE. 2018.
17. Kaur, Jaspreet, and Satwinder Singh. "Neural network based refactoring area identification in software system with object-oriented metrics." Indian Journal of Science and Technology 9.10 (2016): 1-8.
18. Liu, Hui, Zhifeng Xu, and Yanzhen Zou. "Deep learning based feature envy detection." Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018.
19. Xu, Weiwei, and Xiaofang Zhang. "Multi-Granularity Code Smell Detection using Deep Learning Method based on Abstract Syntax Tree."
20. Das, Ananta Kumar, Shikhar Yadav, and Subhasish Dhal. "Detecting code smells using deep learning." TENCON 2019-2019 IEEE Region 10 Conference (TENCON). IEEE, 2019.
21. Saranya, G., et al. "Model level code smell detection using egapso based on similarity measures." Alexandria engineering journal 57.3 (2018): 1631-1642.
22. Sharma, Tushar, et al. "Code smell detection by deep direct-learning and transfer-learning." Journal of Systems and Software 176 (2021): 110936.
23. Guggulothu, Thirupathi, and Salman Abdul Moiz. "Code smell detection using multi-label classification approach." Software Quality Journal 28.3 (2020): 1063-1086.
24. Mansoor, Usman, et al. "Multi-objective code-smells detection using good and bad design examples." Software Quality Journal 25.2 (2017): 529-552.
25. Fontana, Francesca Arcelli, and Marco Zanoni. "Code smell severity classification using machine learning techniques." Knowledge-Based Systems 128 (2017): 43-58.
26. Oliveira, Daniel, et al. "Applying Machine Learning to Customized Smell Detection: A Multi-Project Study." Proceedings of the 34th Brazilian Symposium on Software Engineering. 2020.
27. Caram, Frederico Luiz, et al. "Machine learning techniques for code smells detection: a systematic mapping study." International Journal of Software Engineering and Knowledge Engineering 29.02 (2019): 285-316.
28. Pecorelli, Fabiano, et al. "Comparing heuristic and machine learning approaches for metric-based code smell detection." 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). IEEE, 2019.
29. Draz, Moatasem M., et al. "Code Smell Detection Using Whale Optimization Algorithm." CMC-COMPUTERS MATERIALS & CONTINUA 68.2 (2021): 1919-1935.
30. Tempero, Ewan, et al. "The Qualitas Corpus: A curated collection of Java code for empirical studies." 2010 Asia Pacific Software Engineering Conference. IEEE, 2010.
31. Khleel, Nasraldeen A. A., and Károly Nehéz. "Mining Software Repository: an Overview." Doktoranduszok Fóruma: 188 p. pp. 108-114., 7 p. ISBN: 9789633582121, 2020.
32. Nasraldeen A. A. Khleel and Károly Nehéz, “Comprehensive Study on Machine Learning Techniques for Software Bug Prediction” International Journal of Advanced Computer Science and Applications(IJACSA), 12(8), 2021. http://dx.doi.org/10.14569/IJACSA.2021.0120884