The Journal of Supercomputing (2024) 80:21048–21093
https://doi.org/10.1007/s11227-024-06265-9

# Improving accuracy of code smells detection using machine learning with data balancing techniques

**Nasraldeen Alnor Adam Khleel**[1] · **Károly Nehéz**[1]

Accepted: 21 May 2024 / Published online: 5 June 2024


## Abstract

Code smells indicate 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 software failure risk, and helping to refactor the code. Previous works have applied several prediction methods for code smell detection. However, many of them show that machine learning (ML) and deep learning (DL) techniques are not always suitable for code smell detection due to the problem of imbalanced data. So, data imbalance is the main challenge for ML and DL techniques in detecting code smells. To overcome these challenges, this study aims to present a method for detecting code smell based on DL algorithms (Bidirectional Long Short-Term Memory (Bi-LSTM) and Gated Recurrent Unit (GRU)) combined with data balancing techniques (random oversampling and Tomek links) to mitigate data imbalance issue. To establish the effectiveness of the proposed models, the experiments were conducted on four code smells datasets (God class, data Class, feature envy, and long method) extracted from 74 open-source systems. We compare and evaluate the performance of the models according to seven different performance measures accuracy, precision, recall, *f*-measure, Matthew's 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). After comparing the results obtained by the proposed models on the original and balanced data sets, we found out that the best accuracy of 98% was obtained for the Long method by using both models (Bi-LSTM and GRU) on the original datasets, the best accuracy of 100% was obtained for the long method by using both models (Bi-LSTM and GRU) on the balanced datasets (using random oversampling), and the best accuracy 99% was obtained for the long method by using Bi-LSTM model and 99% was obtained for the data class and Feature envy by using GRU model on the balanced datasets (using Tomek links). The results indicate that the use of data balancing techniques had a positive effect on the predictive accuracy of the models presented. The results show that the proposed models can detect the code smells more accurately and effectively.

Extended author information available on the last page of the article

**Keywords** Code smells · Software metrics · Machine learning (ML) · Deep learning (DL) · Bi-LSTM · GRU · Class imbalance · Data balancing techniques

**Abbreviations**

| | |
|---|---|
| ML | Machine learning |
| Bi-LSTM | Bidirectional long short-term memory |
| GRU | Gated recurrent unit |
| AUC | Area under a receiver operating characteristic curve |
| AUCPR | Area under the precision–recall curve |
| MSE | Mean square error |
| DL | Deep learning |
| CNN | Convolution neural networks |
| RNN | Recurrent neural network |
| BI-RNNs | Bidirectional recurrent neural networks |
| QC | Qualitas corpus |
| FS | Feature selection |

## 1 Introduction

Software development and maintenance involve high costs, especially as systems become more complex. Code smells indicate design or programming issues, which make it difficult to alter and maintain the software [1–3]. Code smells are one of the most accepted approaches to identifying design problems in the source code. Detecting code smells is a significant 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 [4–6], and they showed their adverse effects on the quality of the software.

Previous work provided several tools for code smell detection. These tools rely on detection rules that compare the values of relevant metrics extracted from source code against empirically identified thresholds to discriminate code smells. The limitations of these tools are that the performance is strongly influenced by the thresholds needed to identify smelly and nonsmelly instances. For instance, when the threshold is set too low, it can generate false positives, which are incorrect indications that a part of the code is a code smell. Conversely, when the threshold is set too high, false negatives may occur, which means that valid code smells are not detected by the tool. To overcome these limitations, researchers recently adopted and developed many ML and DL techniques and conducted many experimental studies to detect code smells and obtained different results when applying the same case study, where a classifier is trained on previous releases of the source code by exploiting a set of independent variables (e.g., structural, historical, or textual metrics) [2, 7]. ML and DL have been used in several recent works on code smell detection [8]. DL is a type of ML technique that allows computational models consisting of multiple processing layers to learn to

represent data with multiple levels of abstraction [9]. DL architecture has been widely used to solve many detection, classification, and prediction problems [10, 11]. Long Short-Term Memory (LSTM) and GRU are particular types of Recurrent Neural Networks (RNN) used in DL, designed to recognize patterns in data sequences. LSTM and GRU have been proven particularly effective for problems of classification and detection [10, 12, 13].

When there is an uneven distribution of classes in the training data set, this indicates that this data is imbalanced [14]. Using an imbalanced data set to train classification algorithms can lead to misclassification, as the classifier may be biased and not correctly classify instances of the minority label [13]. Imbalanced class classification biases performance toward the majority numbered class in the case of a binary application [15]. Most ML and DL techniques can predict better when each class's instances are roughly equal. So, data imbalance is the biggest problem faced by ML and DL techniques. This problem severely hinders the efficiency of these techniques and produces unbalanced false-positive and false-negative results [16].

This study selects public benchmark datasets containing 74 open-source systems from Qualitas Corpus for experimental purposes. These datasets are imbalanced, which motivates a solution such as applying data balancing techniques to solve the problem of imbalanced data to help develop an efficient and accurate model for code smell detection. Although several experiments in the previous studies [10, 12, 17, 18] are conducted based on this dataset using many ML and DL algorithms, very few are based on ML and DL algorithms with data balancing techniques.

Data balancing techniques aim to address the class imbalance problem to allow the training of robust and well-fit ML and DL models. Based on studies that have applied data balancing techniques with ML and DL models in detecting code smells [2, 7], the authors indicate a positive effect of data balancing techniques on the ML and DL model's performance. Therefore, this study aims to apply data balancing techniques to address the problem of imbalanced data and investigate the impact of data balancing techniques on the performance of DL models in detecting code smells. Firstly, we apply data balancing techniques to balance the data sets. Secondly, we train and test the proposed models using balanced datasets. Finally, we evaluate the results based on many performance measures: accuracy, precision, recall, *f*-measure, MCC, AUC, AUCPR, and MSE. In summary, the goal and main contributions of our study are summarized as follows:

(i) The paper identifies data imbalance as a major challenge for ML and DL techniques in detecting code smells. This recognition paves the way to propose a novel solution to improve the accuracy of code smell detection.
(ii) To address data imbalance issues and improve the accuracy of code smell detection. This paper aims to present a novel method that combines two DL algorithms (Bi-LSTM and GRU) with two data balancing techniques (random oversampling and Tomek Links) to detect four code smells (God class, Data Class, Feature envy, and Long method).
(iii) To demonstrate the capability, effectiveness, and efficiency of the proposed method for code smell detection, we conducted extensive experiments and

compared the results of the method with the results of state-of-the-art methods in code smell detection based on various performance measures.

The rest of the paper is organized as follows: Sect. 2 discusses related work. Section 3 presents background on code smells, bidirectional LSTM, gated recurrent units, and data balancing techniques. Section 4 presents the hypothesis and research questions. After that, our research methodology is presented in Sect. 5. Section 6 presents the experimental results and discussion, followed by conclusions in the last section.

## 2 Related work

Most approaches for code smell prediction use object-oriented metrics to determine whether a software system contains code smells. Our previous work implemented several ML and DL models combined with various data balancing techniques for code smell detection [2, 13]. Therefore, this study is an extension of our previous works by applying two DL algorithms combined with two data balancing techniques to investigate the role of different data balancing techniques in improving code smell detection.

Various methodologies and techniques, such as classical ML algorithms, neural networks, and DL algorithms, have been proposed in previous work to detect code smells [2, 4, 12, 13, 18–20].

Some approaches [4, 6, 21–23] applied classical ML algorithms to detect code smells. Arcelli Fontana et al. [4] and Mhawish and Gupta [6] presented a method using different ML algorithms and software metrics to detect code smells based on 74 software systems. The experimental results showed that ML techniques have high potential in predicting the code smells, but imbalanced data caused varying performances that need to be addressed in future studies. Dewangan et al. [21] and Jain and Saha [22] proposed a method using several ML algorithms to predict the code smells. The performance of the proposed method was evaluated based on different performance metrics, and two feature selection techniques were applied to enhance the prediction accuracy. The experimental results of Dewangan et al. showed that the best performance was 100%; while, the worst performance was 95.20%. The experimental results of Jain and Saha showed that among several models, boosted decision trees and Naive Bayes gave the best performance after dimensionality reduction. Pontillo et al. [23] proposed a novel test smell detection approach based on six ML to detect four test smells. They assess their models capabilities in within- and cross-project scenarios and compare them with state-of-the-art heuristic-based techniques. The results showed that the proposed approach is significantly better than heuristic-based techniques. Some approaches [2, 7, 13, 18] investigated the role of data balancing methods with ML and DL algorithms in improving the accuracy of code smell detection. Khleel and Nehéz [2, 13] presented various ML and DL algorithms with oversampling methods (random oversampling and synthetic minority oversampling technique (SMOTE)) to detect four code smell (God class, data class, feature envy, and long method). The results were compared based on different

performance measures. Experimental results show that oversampling techniques improve the accuracy of the proposed models and provide better performance for code smell detection. Pecorelli et al. [7] investigated five data balancing techniques to mitigate data imbalance issues to understand their impact on ML algorithms for code smell detection. The experiment was based on five code smell datasets from 13 open-source systems. The experimental results showed that the ML models relying on the synthetic minority oversampling technique realize the best performance. This method solved the class imbalance problem by applying sampling methods. Yet, this method achieved low prediction accuracy in some datasets. Hadj-Kacem and Bouassida [18] proposed a hybrid approach based on deep Autoencoder and artificial neural network algorithms to detect code smells. The approach was evaluated using four code smells extracted from 74 open-source systems. This method solved the class imbalance problem by applying feature selection techniques. The experiment results showed that the recall and precision measurement values were highly accurate. Some approaches [10, 12, 17] applied neural network algorithms to detect code smells. Sharma et al. [10, 12] proposed a new method for code smell detection using Convolution Neural Networks (CNN) and Recurrent Neural Networks (RNN). The experiments were conducted based on C# sample codes. The experiment results showed that it is feasible to detect smells using DL methods, and transfer learning is possible to detect code smells with a performance like that of direct learning. Kaur and Singh [17] suggested a neural network model based on object-oriented metrics for detecting bad code smells. The model was trained and tested using epochs to find twelve bad code smells. The experimental results showed the relationship between bad smells and object-oriented metrics. Some approaches [9, 20, 24] applied the DL algorithm to detect code smells. Liu et al. [9] proposed a new DL-based approach to detecting code smells. The approach was evaluated based on four types of code smell: feature envy, long method, large class, and misplaced class. The experiment results showed that the proposed approach significantly improves the state-of-the-art. Das et al. [20] proposed a DL-based approach to detect two code smells (Brain Class and Brain Method). The proposed system was evaluated using thirty open-source Java projects from GitHub repositories. The experiments demonstrated high accuracy results for both the code smells. Xu and Zhang [24] proposed a novel DL approach based on abstract syntax trees to detect multi-granularity code smells based on four types. Experimental results showed that the approach yields better than the latest methods for detecting code smells with different granularities.

After reviewing previous studies in code smell detection, we noticed that most proposed methods ignore the class imbalance problem. According to studies that dealt with the issue of class imbalance and handled it, the authors point out that the data balancing methods have an essential role in improving the accuracy of code smell detection [2, 7, 13, 18]. So, the primary point from the recent studies is that ML and DL combined with data balancing techniques can improve and increase prediction accuracy. Therefore, our study focuses on solving the class imbalance problem using random oversampling and Tomek Links methods.

## 3 Background

This section presents the background of code smells, Bidirectional LSTM, Gated Recurrent units, and data balancing techniques.

### 3.1 Code smells

Code smells are warning signs which refer to certain patterns or characteristics observed in source code that indicate potential design flaws or violate basic design rules such as abstraction, hierarchy encapsulation, etc. [21, 25]. While code smells are not necessarily indicative of bugs and the code will still work, they can make future development more difficult and increase the risk of bugs and code smells often suggest that the code could be refactored, or changeability of a given part of the source code to improve clarity, maintainability, or efficiency [22–25]. As dependent variables, choose code smells with high frequency that may have the most significant negative impact on the software quality, which can be recognized by some available detection tools [26]. Code smell detection is the primary requirement to guide the subsequent steps in the refactoring process [18, 27]. Detection rules are approaches used to detect code smells through a combination of different software metrics with predefined threshold values. Most current detectors need the specification of thresholds to distinguish smelly and nonsmelly codes [28]. Many approaches have been presented by the authors for uncovering the smells from the software systems. Different detection methodologies differ from manual to visualization-based, semi-automatic studies, automatic studies, empirical-based evaluation, and metrics-based detection of smells. Most techniques used to detect code smells rely on heuristics and discriminate code artifacts affected (or not) by a specific type of smell through the application of detection rules that compare the values of metrics extracted from source code against some empirically identified thresholds. Researchers recently adopted ML and DL to detect code smells to avoid thresholds and decrease the false-positive rate in code smell detection tools [29, 30]. Table 1 lists the four specific code smells that we have investigated.

**Table 1** Lists the four specific code smells that we have investigated [4]

| Code smells | Description | Affected entity |
|---|---|---|
| God Class | Classes have many members and implement different behaviors | Class |
| Data Class | Classes contain only fields (data) and methods for accessing them | Class |
| Feature envy | A 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 |
| Long method | Refers to the too long method. It is classified as a blotter smell that affects method-level entities | Method |

### 3.2 Long short-term memory (LSTM), and bidirectional LSTM

Long Short-Term Memory (LSTM) was introduced to avoid or handle long-term dependency problems without being affected by an unstable gradient [10]. This problem frequently occurs in regular RNNs when connecting previous information to new information. A standard LSTM unit comprises a cell, an input gate, an output gate, and a forget gate. The cell remembers values over arbitrary time intervals, and the three gates regulate the flow of information into and out of the cell. Due to the ability of the LSTM network to recognize longer sequences of time-series data, LSTM models can provide high predictive performance in code smell detection. More recently, Bidirectional long-short-term memory (Bi-LSTM) is a new way to train data by expanding the capabilities of LSTM networks; it uses two separate hidden layers to train the input data twice in the forward and backward directions, as shown in Fig. 1. With the regular LSTM, the input flows in one direction, either backward or forward. Bi-LSTM is a process of making any neural network have the sequence information in both directions (a sequence processing model that consists of two LSTMs): one taking the input in a forward direction (past to future) and the other in a backward direction (future to past). The idea behind Bi-LSTM is to exploit spatial features to capture bidirectional temporal dependencies from historical data to overcome the limitations of traditional RNNs [28]. Standard RNN takes the sequences as inputs, and each sequence step refers to a certain moment [10, 12]. 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 ($t$).

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

where $U$, $V$, and $W$ denote the weights of the RNN, $b$, and $c$ represent 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

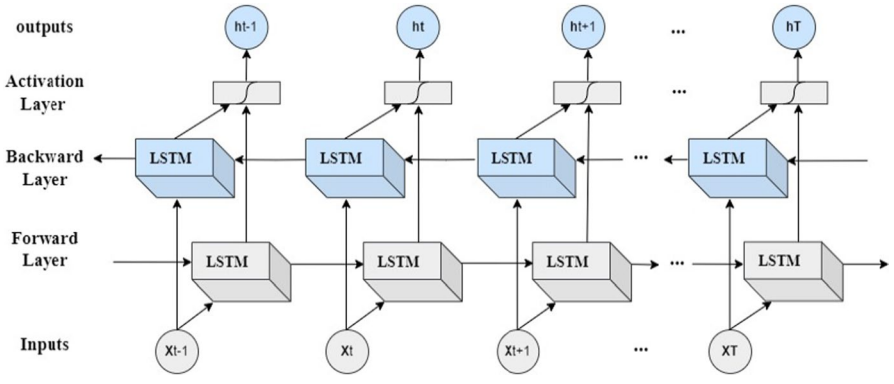


**Fig. 1** Interacting layers of the repeating module in a Bi-LSTM "Figure adapted from Verma [31]"

the key that LSTM can have long-should be filtered of the prior moment; the output of forget gate can be formulated as the following equation:

$$f_t = \sigma\left(W_{\mathrm{f}} \cdot \left[h_{t-1,} x_t\right] + b_{\mathrm{f}}\right) \tag{2}$$

where σ denotes the activation function, $W_{\mathrm{f}}$ and $b_{\mathrm{f}}$ 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:

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

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 the forget gate and input gate, the cell state $C_{t-1}$ is updated through the following formula:

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

$\check{C}_t$ is a candidate value that is going to be added to 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.

$$\begin{aligned} o_t &= \sigma(W_o \cdot \left[h_{t-1,} x_t + b_o\right] \\ h_t &= o_t \times \tanh\left(C_t\right). \end{aligned} \tag{5}$$

### 3.3 Gated recurrent unit

GRU network is one of the optimized structures of the RNN. 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 the learning sequence has a gradient vanishing and gradient explosion phenomenon. Many optimization theories and improved algorithms have been introduced to solve this problem, such as LSTM, GRU networks, Bi-LSTM, echo state networks, and independent RNNs [12]. The GRU network aims 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 Fig. 2 [28]. 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 network is calculated as shown in the equation below.

$$z(t) = \sigma(W(z) \cdot [h(t-1), x(t)]) \tag{6}$$

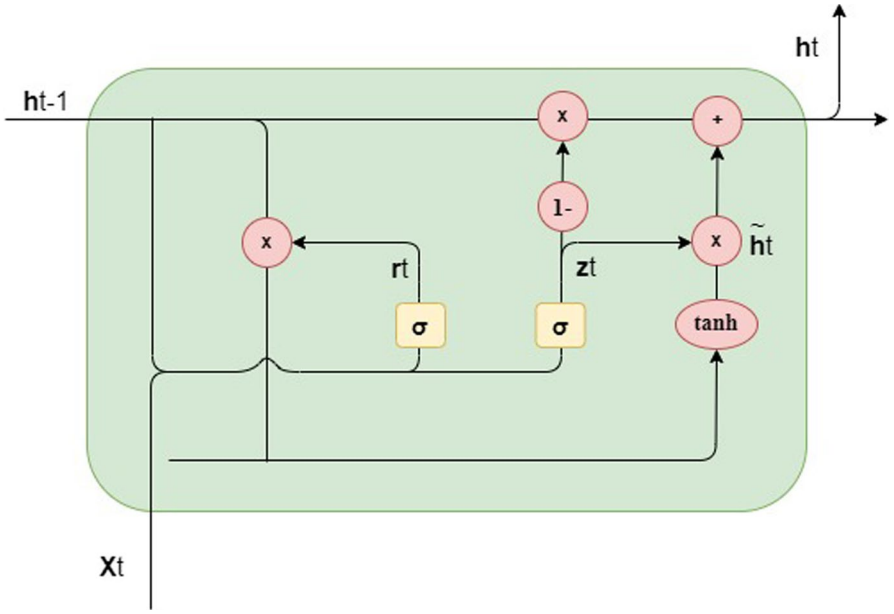


**Fig. 2** Interacting layers of the repeating module in GRU "Figure adapted from Christopher [32]."

$z(t)$ 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 $\sigma$ represents the sigmoid function. The reset gate model in the GRU neural networks is calculated as shown in the equation below.

$$r(t) = \sigma(W(r) \cdot [h(t-1), x(t)]) \tag{7}$$

$r(t)$ 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 $\sigma$ represents the sigmoid function. The output value of the GRU hidden layer is shown in the equation below.

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

$\check{h}(t)$ 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 final output is shown in the equation below.

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

### 3.4 Data balancing techniques

Most code smells datasets are imbalanced, meaning that the target group (smelly classes) has a lower distribution than the nonsmelly classes. Classifying this data type is one of the most challenging problems facing ML and DL algorithms [2, 15]. Several data balancing techniques have been developed to overcome the class imbalance problem, including subset methods, cost-sensitive learning, algorithm-level implementations, ensemble learning, feature selection methods, sampling methods, etc. [16]. The most common techniques used in previous work to deal with the class imbalance problem are sampling methods (oversampling and under-sampling methods) [8]. Sampling methods tend to adjust the prior distribution of the majority and minority classes in the training data to obtain a balanced class distribution [2, 15]. The oversampling method supplements instances of the minority class to the dataset; while, the under-sampling method eliminates samples of the majority class to obtain a balanced dataset; despite its simplicity, random under-sampling is one of the most effective resampling methods [2, 16]. Oversampling methods are more effective than under-sampling methods in prediction accuracy [2, 7]. Random oversampling is a method developed to increase the size of a training data set by making multiple copies of some minority classes; Tomek Links is an under-sampling method developed to randomly select samples with its $k$-nearest neighbors from the majority class that wants to be removed [15].

## 4 Hypothesis and research questions

Our hypothesis in this study is that if data balancing techniques are applied to balance the original data sets, the classification performance of the proposed models will be better in detecting code smells. To investigate our hypothesis, we used a paired $t$-test to determine whether there was a statistically significant difference between our models on the original and balanced datasets, and between our models on the balanced datasets and existing approaches based on the accuracy values. The formula for the paired $t$-test is shown in Eq. 10 below. To statistically prove the validity of the impact of data balancing techniques on the performance of DL algorithms, the hypothesis is formed as follows:

**H1** There is no difference in the accuracy of models when there are no data balancing techniques and when the data balancing techniques are used.

**H2** There is a difference in models' accuracy when there are no data balancing techniques and when the data balancing techniques are used.

$$t = \frac{m}{s/\sqrt{n}} \tag{10}$$

where $m$ is the mean differences, $n$ is the sample size (i.e., number of pairs), and $s$ is the standard deviation.

Based on our hypothesis, this study aims to understand the impact of data balancing techniques on the performance of DL algorithms in detecting code smells. In particular, we aim to address the following research questions.

**RQ1** Do data balancing techniques improve DL models' accuracy in detecting code smells?

This RQ aims to investigate data balancing techniques in improving DL models' accuracy in detecting code smells.

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

This RQ aims to verify the best data balancing technique for detecting code smells.

**RQ3** Does the proposed method outperform the state-of-the-art methods in detecting code smells?

This RQ aims to investigate the performance of the proposed method in detecting code smells compared to the state-of-the-art methods.

## 5 Proposed methodology

The main objective of this study is to perform an empirical study on the role of data balancing techniques in improving DL accuracy for detecting code smells. A series of steps have been taken and described, such as data modeling and collection, data pre-processing, features selection, class imbalance, model building, and evaluation. Figure 3 illustrates the overview of the proposed methodology for code smell detection, where each step is described in the following sections.

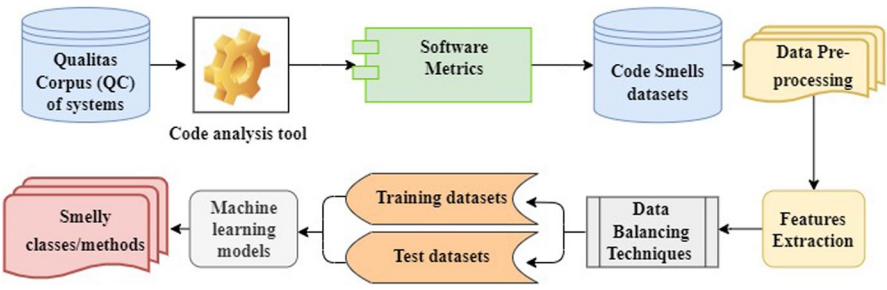


**Fig. 3** Overview of the proposed methodology for code smells detection

### 5.1 Data modeling and collection

Dataset selection is an essential task in the problem of ML and DL, and classification models perform better if the dataset is more relevant to the problem. This study's code smell detection models use a supervised learning task that relies on a large set of software metrics as independent variables. Many systems or datasets are fundamental to training ML and DL models, and allowing the generalization of the results. We considered the Qualitas Corpus (QC) of systems collected by Tempero et al. [33] to perform the analysis. The QC systems comprise 111 systems written in Java, characterized by different sizes and belonging to different application domains, as shown in Table 2. The authors utilized a tool called Design Features and Metrics for Java (DFMC4J) to parse the source code of Java projects through the Eclipse JDT library and calculate all relevant metrics necessary to define code smells. The reason for selecting these datasets is that (1) 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. (2) 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, encapsulation, coupling, and inheritance [4].

### 5.2 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 help identify patterns and indicators associated with software bugs or code smells [34]. Software metrics are widely used indicators of software quality and many studies have shown that these metrics can be used to estimate the presence of vulnerabilities in code [35]. Furthermore, Researchers have shown that software metrics can be effectively used to assess software reusability [36]. Software metrics can be classified as static code metrics and process metrics. Static code metrics can be directly extracted from 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 code smell. The first use of McCabe metrics was to characterize code features related to software quality.

**Table 2** Summary of project characteristics [4]

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

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 [4]. The computed metrics for all 74 systems of the QC are displayed in Table 3.

### 5.3 Data pre-processing

Pre-processing the collected data is one of the critical stages before constructing the model. To generate a good model, data quality needs to be considered [2]. Not all data collected is suitable for training and model building. The inputs will significantly impact the model's performance and later affect the output. Data pre-processing is known as a group of techniques that are applied to the data to improve the quality of the data before model building to remove noise and unwanted outliers from the data set, deal with missing values, feature type conversion, etc. The data set used in this study is a clean copy [2, 6]. Normalization is necessary to convert the values into scaled values (scaling of the data in numeric variables in the range of 0 to 1) to increase the model's efficiency [6]. Therefore, the data set was normalized using Min–Max normalization based on the Python framework (TensorFlow). The formula for calculating the normalized score can be described in (11).

$$x_i = \left(x_i - X\min\right)/(X\max - X\min) \tag{11}$$

where $\max(x)$ and $\min(x)$ represent the maximum and minimum value of the attribute x, respectively.

### 5.4 Feature selection

Feature Selection (FS) is a crucial step in selecting the most discriminative features from the list of features using appropriate FS methods [37]. FS aims to choose the features more relevant to the target class from high-dimensional features and remove the redundant and uncorrelated features [38]. Feature extraction facilitates the conversion of pre-processed data into a form that the classification engine can use [39]. FS methods are categorized into three categories (i) Filter methods: These methods are model agnostic, i.e., variables are selected independently of ML or DL algorithms. These methods are faster and less computationally expensive, (ii) Wrapper methods: These methods are greedy and choose the best feature subsets in each iteration according to ML or DL algorithms. It is a continuous process of finding a feature subset. These methods are very computationally expensive and often unrealistic if the feature space is vast, (iii) Embedded methods: FS is part of building ML or DL algorithms in these methods, where feature selection is integrated into the model training process itself [22]. Unlike filter methods that pre-process data before model training or wrapper methods that select features based on the performance of a specific model, embedded methods embed feature selection directly within the model training process. These methods select the best possible feature subset as per the ML or DL model to be implemented [40]. In this study, we applied embedded methods

**Table 3** Show the computed metrics for all 74 systems of the QC [4]

| Size | Complexity | Cohesion | Coupling | Encapsulation | Inheritance |
|---|---|---|---|---|---|
| Lines of Code (LOC) | McCabe's Cyclomatic complexity (CYCLO) | Lack of cohesion between methods (LCOM) | Class fan out complexity (CLASS_FAN_OUT) | Locality of attribute accesses (LAA) | Depth of inheritance tree (DIT) |
| Lines of code excluding accessor and mutator methods (LOCNAMM*) | Weighted methods per class (WMC) | Tight class cohesion (TCC) | Access to foreign data (ATFD) | Number of Accessor methods (NOAM) | Response for a class (RFC) |
| Number of methods (NOM) | Weighted methods count of not accessor or mutator methods (WMC-NAMM*) | | Foreign data providers (FDP) | Number of public attributes (NOPA) | Number of children (NOC) |
| Number of packages (NOPK) | Average methods weight of not accessor or mutator methods (AMWNAMM*) | | Coupling between objects (CBO) | | Number of methods overridden (NMO) |
| Number of classes (NOCS) | Average methods weight (AMW) | | Called foreign not accessor or mutator methods (CFNAMM*) | | Number of inherited methods (NIM) |
| Number of methods excluding accessor and mutator methods (NOM-NAMM*) | Maximum nesting level of control structures (MAXNESTING) | | Coupling intensity (CINT) | | Number of implemented interfaces (NOII) |
| Number of attributes (NOA) | Weight of class (WOC) | | Coupling dispersion (CDISP) | | |
| | Called local not accessor or mutator methods (CLNAMM) | | Maximum message chain length (MaMCL§) | | |
| | Number of parameters (NOP) | | Mean message chain length (MeMCL§) | | |

**Table 3** (continued)

| Size | Complexity | Cohesion | Coupling | Encapsulation | Inheritance |
|---|---|---|---|---|---|
| | Number of access variables (NOAV) | | Number of message chain statements (NMCS§) | | |
| | Access to local data (ATLD*) | | Control coupling (CC) | | |
| | Number of local variable (NOLV) | | Number of methods affected by the measured method (CM) | | |

because it is faster and less computationally expensive than other methods. It fits ML and DL models, and a feature scaling technique was used to make the output the same standard.

### 5.5 Class imbalance

Class imbalance in classification models represents those situations where the number of examples of one class is much smaller than another. The class imbalance problem makes classification models not effectively predict minority modules [2, 13]. This study's dataset chosen for code smell detection is highly imbalanced [4]. The original datasets were composed of 561 smelly instances and 1119 nonsmelly instances; the two first datasets concern the code smells at the class level, for God Class (the number of smelly instances is 140, and the number of nonsmelly instances is 280), for Data Class (the number of smelly instances is 140 and the number of nonsmelly instances is 280). The two-second datasets concern the code smells at the method level, for Feature Envy (the number of smelly instances is 140 and the number of nonsmelly instances is 280), for Long Method (the number of smelly instances is 141 and the number of nonsmelly instances is 279). To solve the problem of class imbalance and increase the realism of the data, we modified the original datasets by changing the distribution of instances with the algorithms of Random Oversampling and Tomek Links. After balancing the data sets using the random oversampling method, for God Class (the number of smelly instances is 280 and the number of nonsmelly instances is 280), for Data Class (the number of smelly instances is 280 and the number of nonsmelly instances is 280), for Feature Envy (the number of smelly instances is 280 and the number of nonsmelly instances is 280), for Long Method (the number of smelly instances is 279 and the number of nonsmelly instances is 279). After balancing the data sets using the Tomek links method, for God Class (the number of smelly instances is 140 and the number of nonsmelly instances is 263), for Data Class (the number of smelly instances is 140 and the number of nonsmelly instances is 256), for Feature Envy (the number of smelly instances is 140 and the number of nonsmelly instances is 261), for Long Method (the number of smelly instances is 141 and the number of nonsmelly instances is 270). Figures 4 and 5 show the distribution of learning instances over original and balanced datasets.

### 5.6 Models building and evaluation

Different ML and DL algorithms are used to build code smell prediction models, and each algorithm has its benefits [2, 12]. An empirical study has been conducted to demonstrate the effectiveness and accuracy of the proposed DL models for code smell detection. The proposed models have been trained and tested using a set of massive open-source Java projects to get more accurate results. The choice of paradigm, or framework within a given programming language can significantly impact the development process, development speed, code complexity, and compatibility with other systems [3]. Therefore, after an extensive review of the

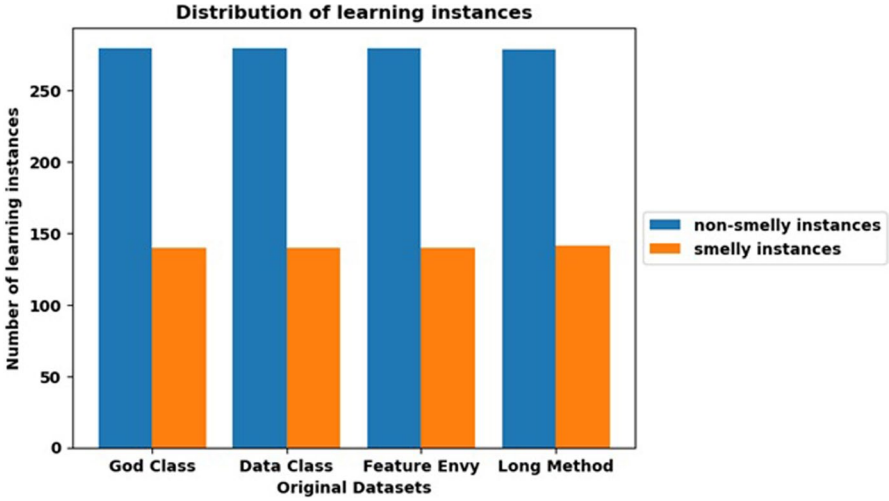


**Fig. 4** Distribution of learning instances over the original datasets

existing literature and leveraging the collective knowledge of the field, the decision to utilize a dynamically typed language like Python, coupled with a robust open-DL framework such as TensorFlow, emerges as a prudent and advantageous choice for building and training DL models [2]. In the realm of implementation frameworks, we use Keras as a high-level API based on TensorFlow to build our models for simplicity and correctness; training is performed with 80% of the dataset (random selection of features), while the remaining 20% is used for validation and testing. The architecture of the proposed LSTM and GRU models consists of several components, each playing a crucial role in the model's functionality. Firstly, the input phase involves utilizing software metrics as inputs, which serve as the foundational data for the subsequent layers to analyze. Secondly, the LSTM and GRU Layers are pivotal elements, employing multiple layers of both LSTM and GRU units. These layers are designed to adeptly capture sequential dependencies within the code, facilitating the model's ability to understand and learn from intricate patterns and long-range dependencies. The third component, the Output Layer, is comprised of a dense output layer with sigmoid activation, responsible for producing the final predictions based on the learned features. Moving on to the fourth component, the Loss Function and Optimization phase, the MSE is employed as the Loss Function; while, the Adam optimizer is utilized for efficient parameter updating during training. This combination ensures that the model is effectively trained to minimize errors and enhance predictive accuracy. Additionally, the fifth component, Hyperparameter Tuning, plays a critical role in optimizing model performance and generalization. Various hyperparameters such as learning rate, batch size, number of layers, hidden units, and dropout regularization between LSTM and GRU layers are meticulously tuned. This tuning process is essential for preventing overfitting, optimizing the model's performance, and enhancing its ability to generalize to unseen data. It's noteworthy that each model

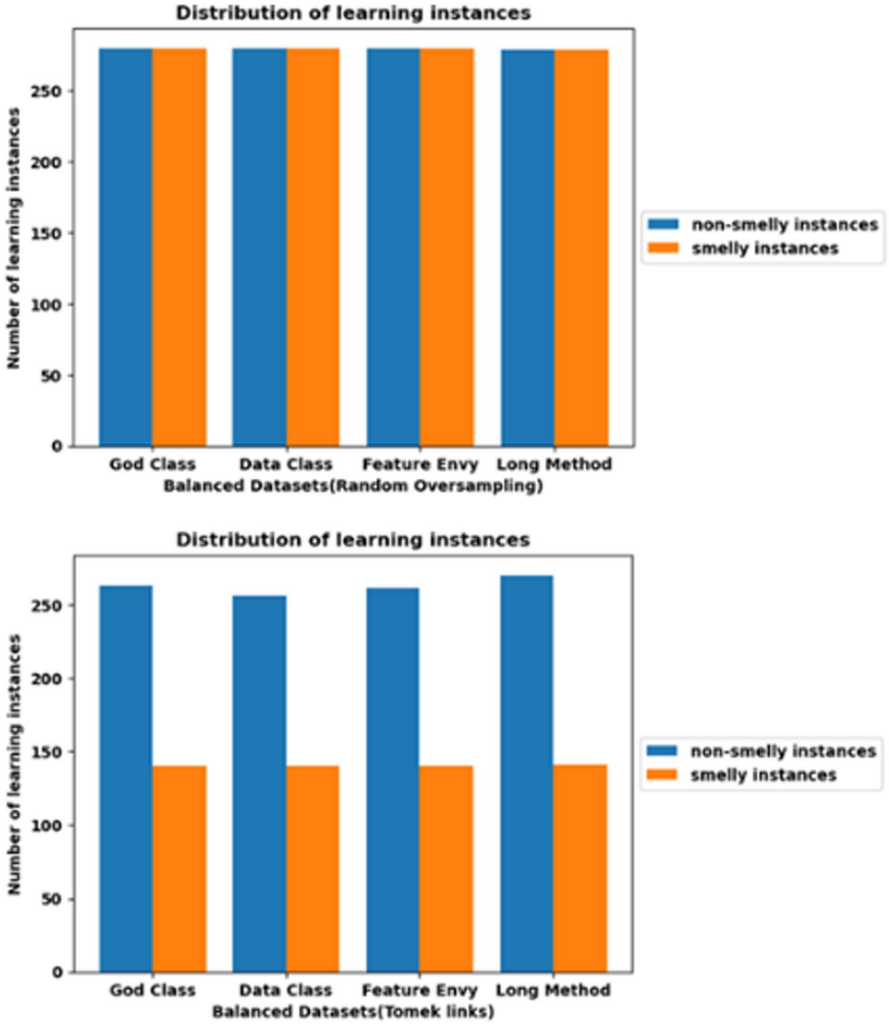


**Fig. 5** Distribution of learning instances over the balanced datasets

was developed separately as shown in Table 4. Moreover, Cross-validation is a vital technique in ML or DL 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 [5, 39]. Cross-validation helps mitigate issues like overfitting, and tuning hyperparameters, selecting the best model, and provides a more reliable assessment of how well a model will perform on unseen data 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

**Table 4** Parameters setting of the models

| Parameters | Models | |
|---|---|---|
| | Bi-LSTM | GRU |
| Cell type (the type of recurrent unit used within each RNN cell) | LSTM | GRU |
| Activation function (a function that is used to get the output of a node or a layer of neurons) | ReLU + Sigmoid | Tanh + Sigmoid |
| Dropouts (Dropout is a regularization technique used to prevent overfitting and improve the generalization performance of the model) | 0.2 | 0.2 |
| Dense (Dense is a layer used to connect every neuron in the preceding layer to every neuron in the subsequent layer.) | 64, 1 | 1 |
| Optimizer (An optimizer is an algorithm used in neural network training and is responsible for updating the parameters (weights and biases) of the model to minimize the loss function and improve the model's performance on the training data) | Adam | Adam |
| Learning Rate (The learning rate is a tuning parameter in an optimization algorithm that determines the step size at each iteration) | 0.01 | 0.01 |
| Loss Function (Loss function is a function used to measure the difference between actual and training outputs.) | Mean Squared Error"MSE" | Mean Squared Error"MSE" |
| Batch_Size (The batch size refers to the number of training examples utilized in one iteration of gradient descent during the training process of a neural network) | 64 | 64 |
| Epochs (refers to one complete pass through the entire training dataset) | 100 | 100 |
| Validation_Split (The validation split is a technique used to evaluate the model's performance and prevent overfitting) | 0.1 | 0.1 |
| Verbose (refers to a setting or parameter that controls the amount of information printed or displayed during the training process) | 1 | 1 |

different data characteristics and modeling objectives. *K*-Fold Cross-Validation and Stratified *K*-Fold Cross-Validation are the most standard methods of Cross-validation [21, 38]. 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 [38]. Therefore, we applied the Stratified *K*-Fold Cross-Validation method to evaluate the performance of our proposed predictive models. This study used a set of standard performance measures based on the confusion matrixes, MCC, AUC, AUCPR, and MSE as Loss Function.

A confusion matrix is a specific table used to measure the performance of a model [6]. A confusion matrix 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). From the values in the confusion matrix, various performance metrics can be derived, such as accuracy, precision, recall, and F1-measure shown in the below equations. These metrics provide insights into the model's strengths and weaknesses, especially in scenarios where class imbalance is present [6, 13].

$$\text{Accuracy} = (\text{TP} + \text{TN})/(\text{TP} + \text{FP} + \text{FN} + \text{TN}) \tag{12}$$

$$\text{Precision} = \text{TP}/(\text{TP} + \text{FP}) \tag{13}$$

$$\text{Recall} = \text{TP}/(\text{TP} + \text{FN}) \tag{14}$$

$$\text{F - Measure} = (2 * \text{Recall} * \text{Precision})/(\text{Recall} + \text{Precision}) \tag{15}$$

The MCC is a performance metric for binary classification. MCC is used for model evaluation by measuring the difference and describing the correlation between the predicted and actual values [2]. The MCC formula is shown in the equation below:

$$\text{MCC} = \text{TP}*\text{TN} - \text{FP}*\text{FN}/\sqrt{(\text{TP} + \text{FP})*(\text{TP} + \text{FN})*(\text{TN} + \text{FP})*(\text{TN} + \text{FN})} \tag{16}$$

AUC graph shows the performance of classification models with all classification thresholds and plots based on two parameters, actual positive rate (TPR.) and false-positive rate (FPR.) [22]. The AUC formula is shown in the equation below:

$$\text{AUC} = \frac{\sum_{\text{ins}_i \in \text{Positive Class}} \text{rank}\left(\text{ins}_i\right) - \frac{M(M+1)}{2}}{M \cdot N} \tag{17}$$

where $\sum_{\text{ins}_i \in \text{Positive Class}} \text{rank}\left(\text{ins}_i\right)$ it is the sum of the ranks of all positive samples, and M and N are the numbers of positive and negative samples, respectively.

AUCPR is a curve that plots the Precision versus the Recall or a single number summary of the information in the precision–recall curve [2]. The AUCPR formula is shown in the equation below:

$$\text{AUCPR} = \int_0^1 \text{Precision}(\text{Recall})d(\text{Recall}) \tag{18}$$

MSE is a metric that measures the amount of error in the model. It assesses the average squared difference between the actual and predicted values. The MSE formula is shown in the equation below:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(x(i) - y(i))^2 \tag{19}$$

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

## 6 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 a Python environment. The performance of the prediction models on the original and balanced datasets is reported in Tables 5, 6, 7 and 8, and Figs. 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, and 23.

Table 5 presents the results of our 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

**Table 5** Evaluation results for the original datasets

| Datasets | Performance measures | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Accuracy | Precision | Recall | *F*-measure | MCC | AUC | AUCPR | MSE |
| *Bi-LSTM model* | | | | | | | | |
| 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* | | | | | | | | |
| 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 6** Evaluation results for the balanced datasets—random oversampling

| Datasets | Performance measures | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Accuracy | Precision | Recall | F- measure | MCC | AUC | AUCPR | MSE |
| *Bi-LSTM model* | | | | | | | | |
| 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* | | | | | | | | |
| God class | 0.96 | 0.95 | 0.98 | 0.97 | 0.92 | 0.96 | 0.93 | 0.033 |
| Data class | 0.98 | 0.98 | 0.98 | 0.98 | 0.96 | 0.99 | 0.99 | 0.023 |
| Feature envy | 0.97 | 0.95 | 1.00 | 0.98 | 0.94 | 0.97 | 0.95 | 0.032 |
| Long method | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.002 |
| Averages | 0.97 | 0.97 | 0.99 | 0.98 | 0.95 | 0.98 | 0.96 | 0.022 |

**Table 7** Evaluation results for the balanced datasets—Tomek links

| Datasets | Performance measures | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Accuracy | Precision | Recall | F- measure | MCC | AUC | AUCPR | MSE |
| *Bi-LSTM model* | | | | | | | | |
| 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* | | | | | | | | |
| 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 |

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 8** Confusion matrix of our best model (GRU Model) on balanced datasets using Tomek links

| Datasets | Predicted | Actual | |
|---|---|---|---|
| | | Non-smelly instances | Smelly instances |
| God class | Non-smelly instances | TN (58) | FP (3) |
| | Smelly instances | FN (0) | TP (20) |
| Data class | Non-smelly instances | TN (56) | FP (0) |
| | Smelly instances | FN (1) | TP (23) |
| Feature envy | Non-smelly instances | TN (51) | FP (0) |
| | Smelly instances | FN (1) | TP (29) |
| Long method | Non-smelly instances | TN (52) | FP (0) |
| | Smelly instances | FN (2) | TP (29) |

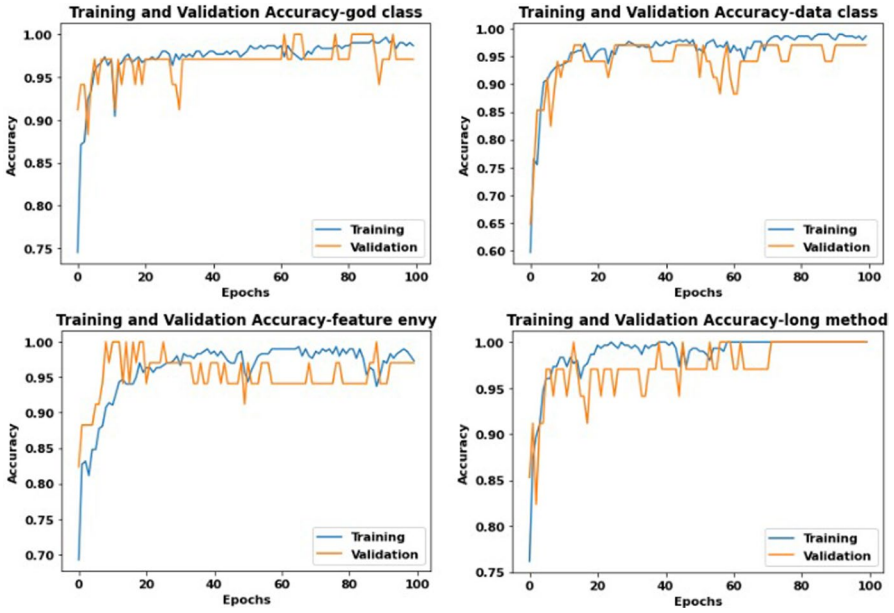


**Fig. 6** Training and validation accuracy on the original datasets using the Bi-LSTM model

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, 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

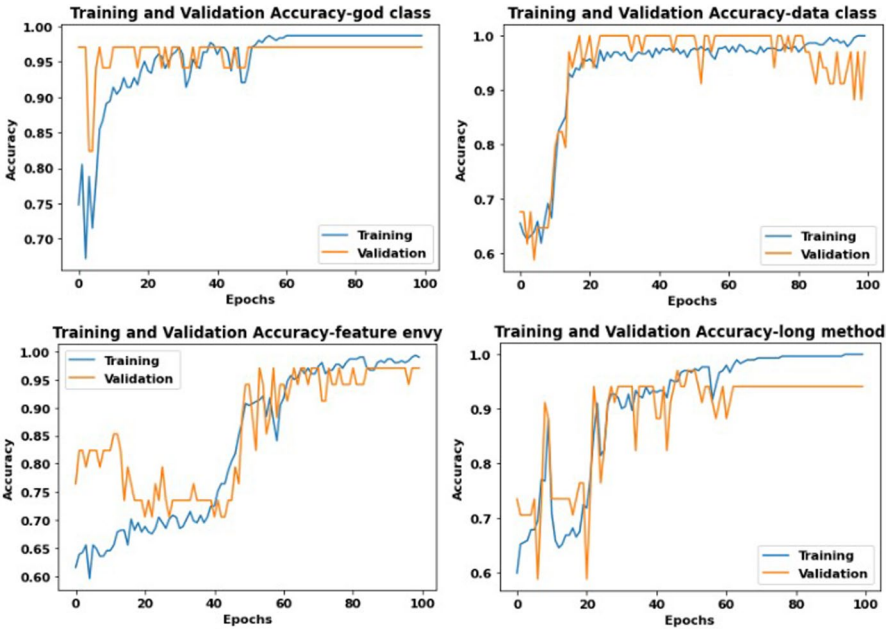


**Fig. 7** Training and validation accuracy on the original datasets using the GRU model

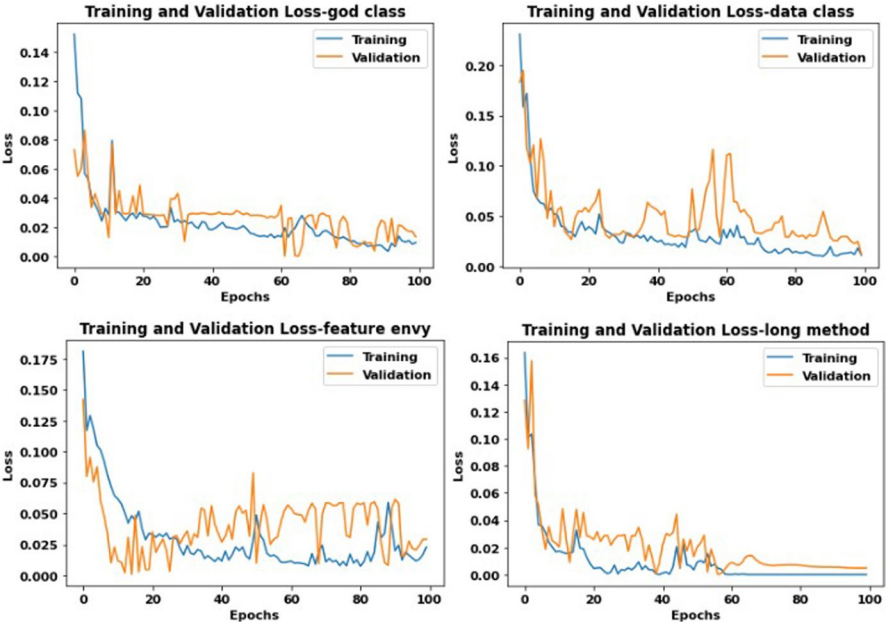


**Fig. 8** Training and validation loss on the original datasets using the Bi-LSTM model

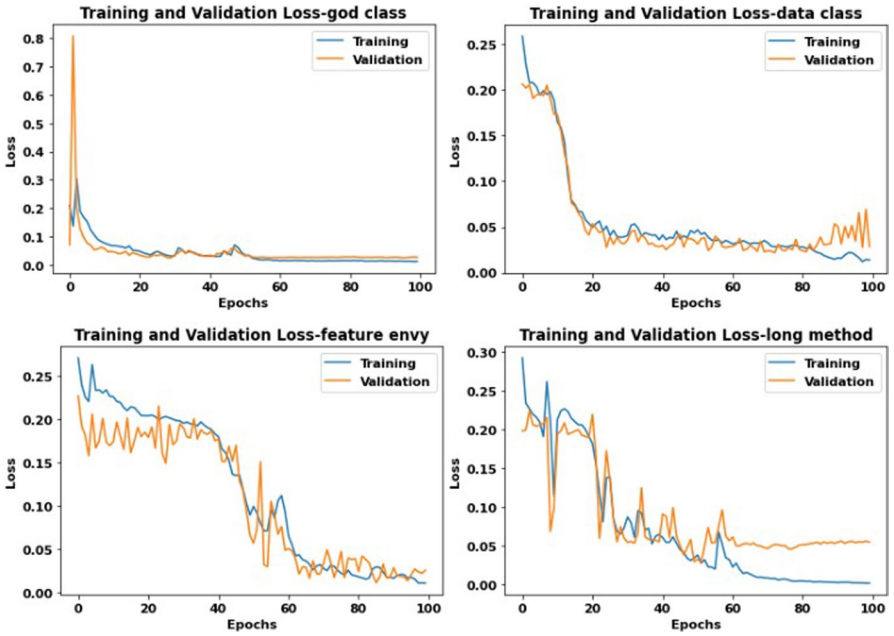


**Fig. 9** Training and validation loss on the original datasets using the GRU model

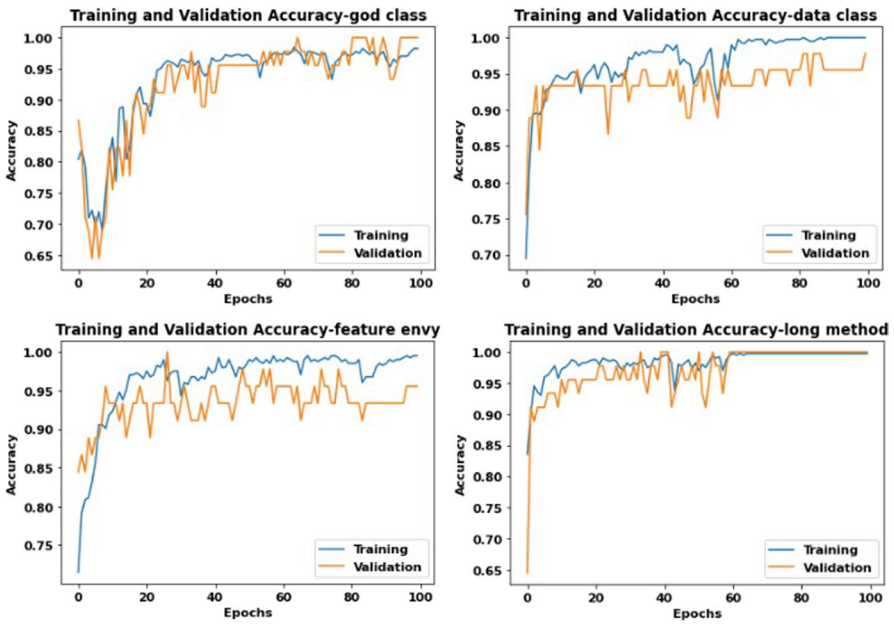


**Fig. 10** Training and validation accuracy on the balanced datasets using Bi-LSTM model-random oversampling

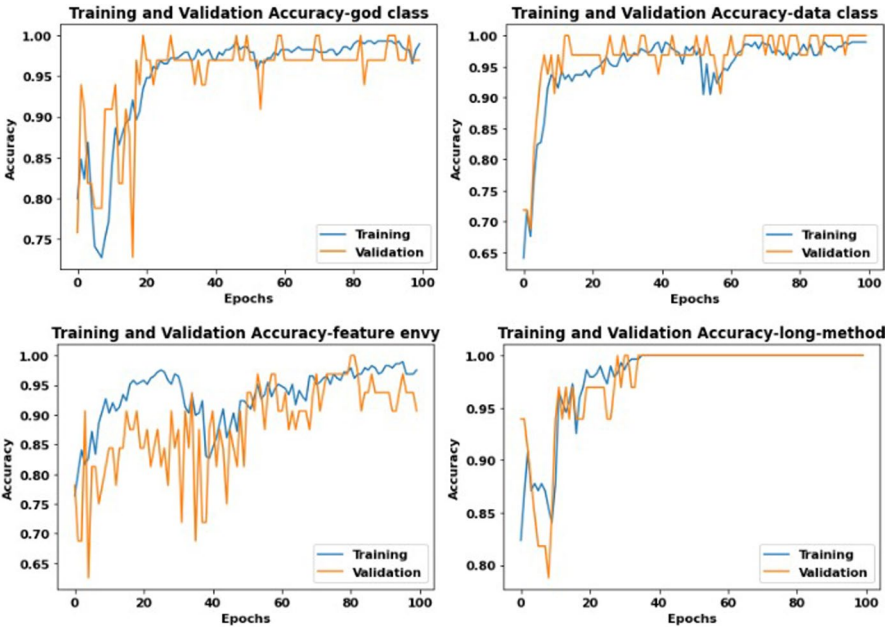


**Fig. 11** Training and validation accuracy on the balanced datasets using Bi-LSTM model-Tomek links

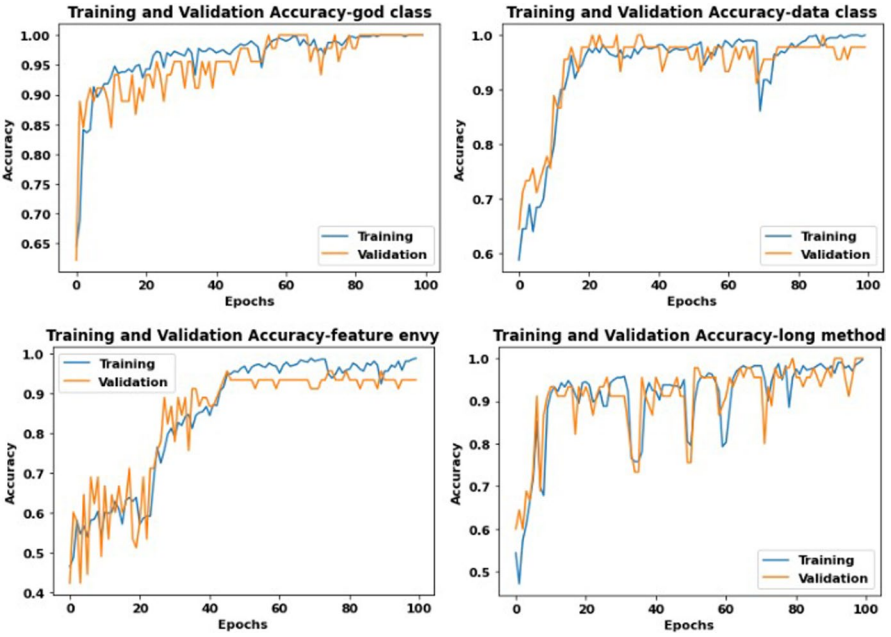


**Fig. 12** Training and validation accuracy on the balanced datasets using GRU model-random oversampling

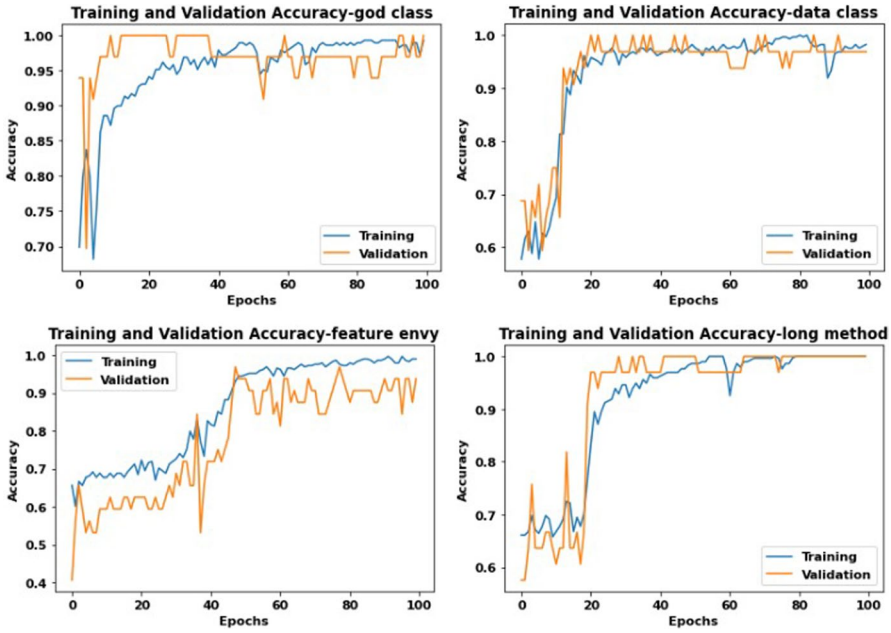


**Fig. 13** Training and validation accuracy on the balanced datasets using GRU model-Tomek links

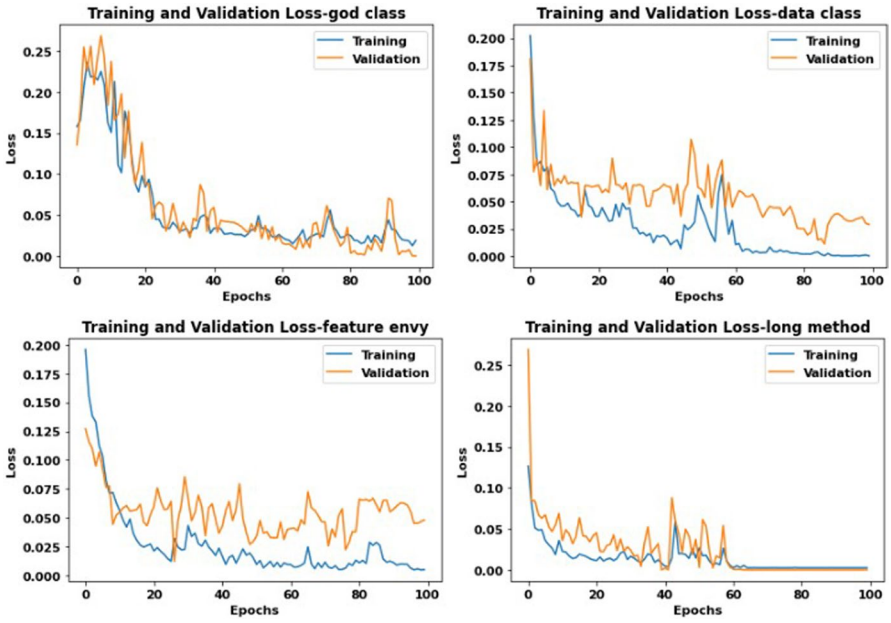


**Fig. 14** Training and validation loss on the balanced datasets using Bi-LSTM model-random Oversampling

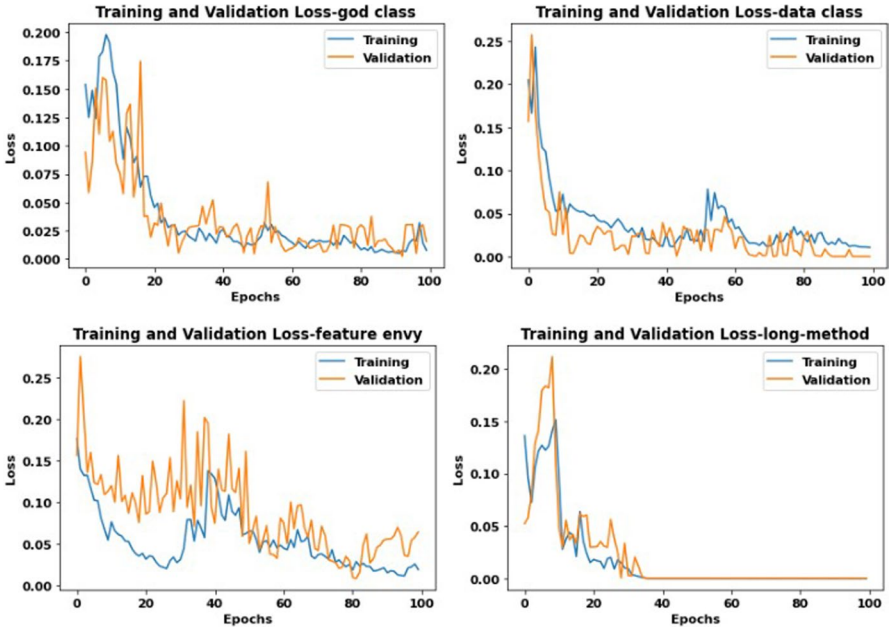


**Fig. 15** Training and validation loss on the balanced datasets using Bi-LSTM model-Tomek links

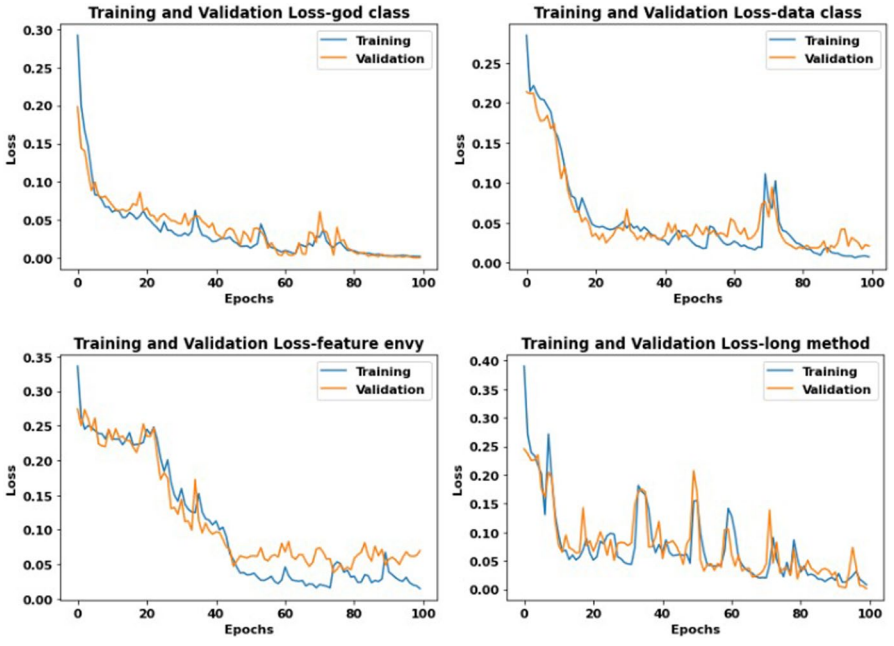


**Fig. 16** Training and validation loss on the balanced datasets using GRU model-random oversampling

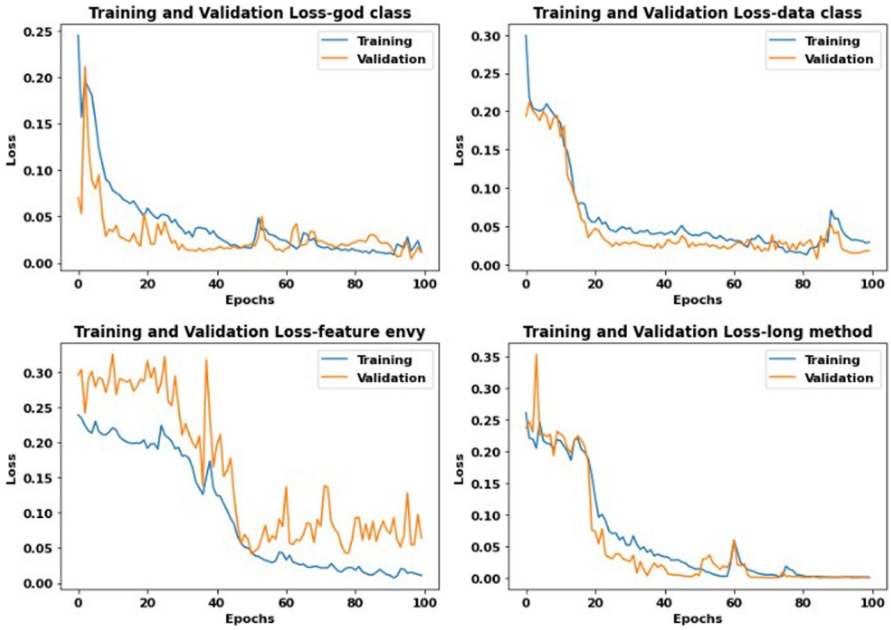


**Fig. 17** Training and validation loss on the balanced datasets using GRU model-Tomek links

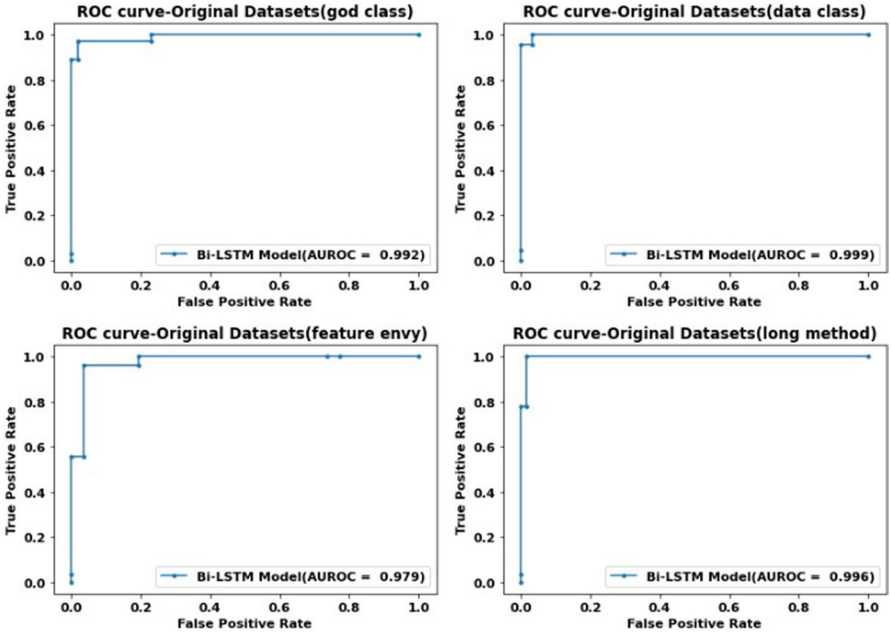


**Fig. 18** ROC curves for the original datasets—Bi-LSTM model

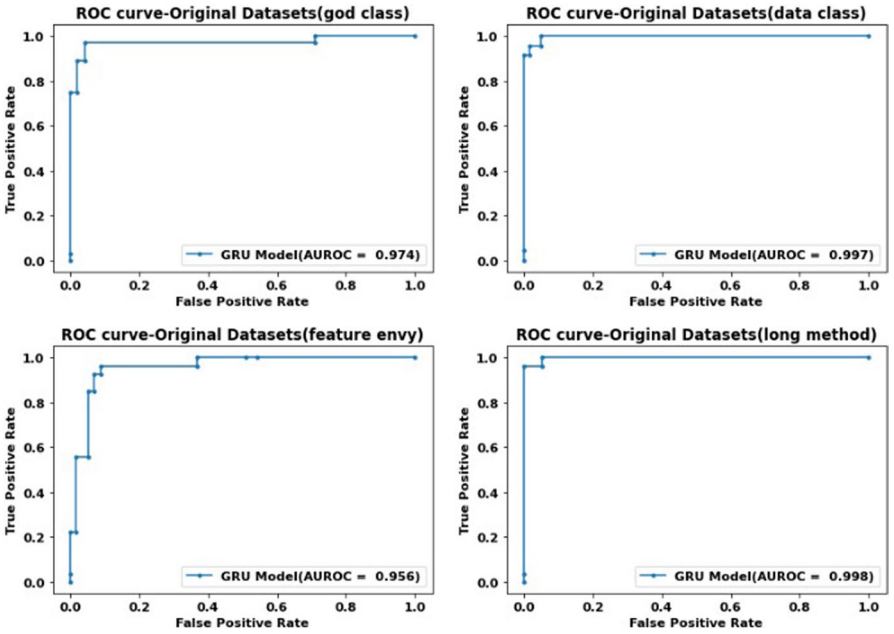


**Fig. 19** ROC curves for the original datasets—GRU model

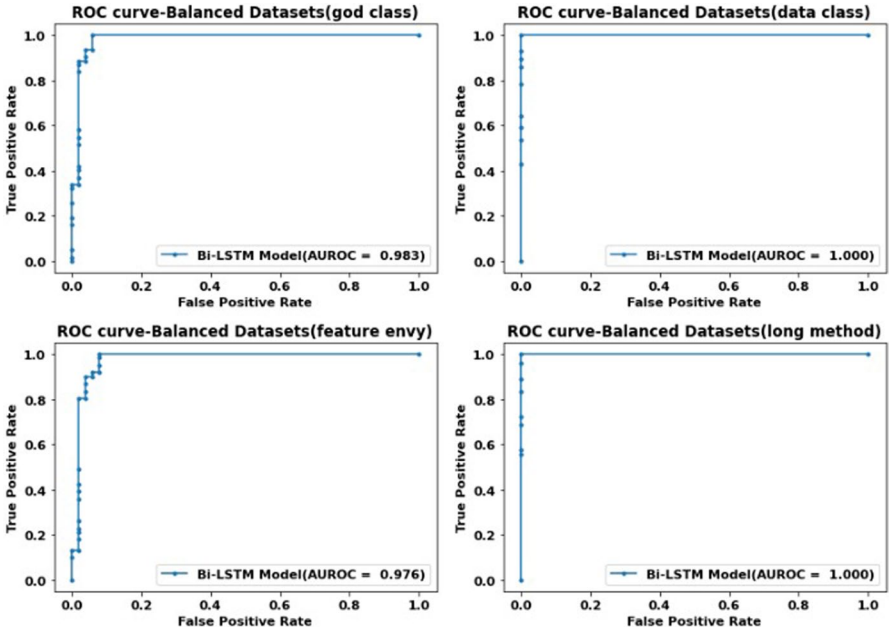


**Fig. 20** ROC curves for the balanced datasets—Bi-LSTM model-random oversampling

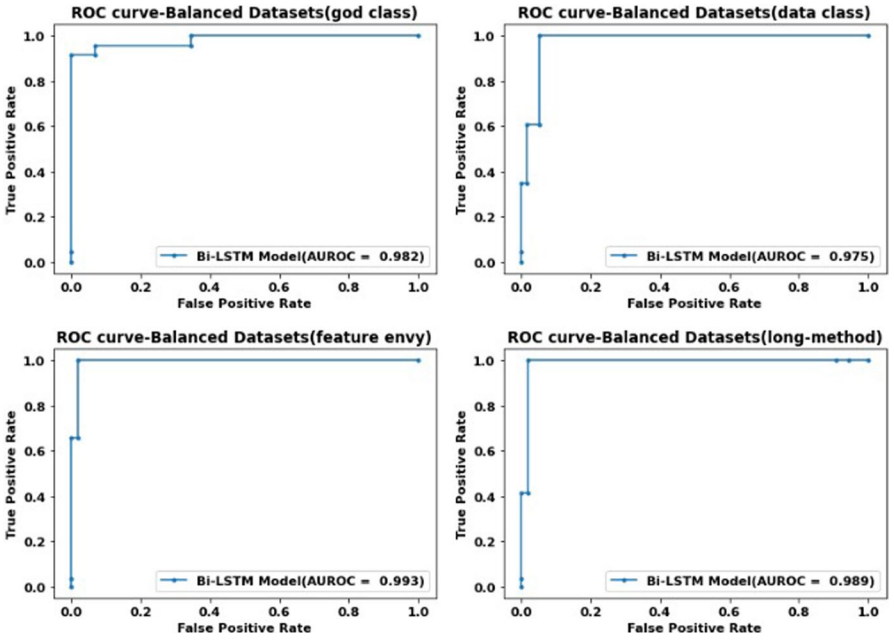


**Fig. 21** ROC curves for the balanced datasets—Bi-LSTM model-Tomek links

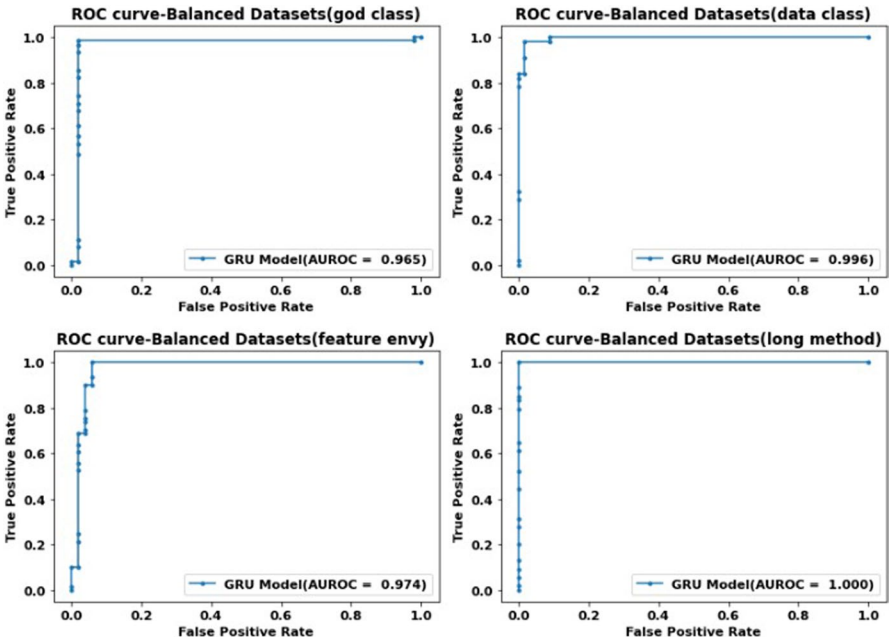


**Fig. 22** ROC curves for the balanced datasets—GRU Model-random oversampling

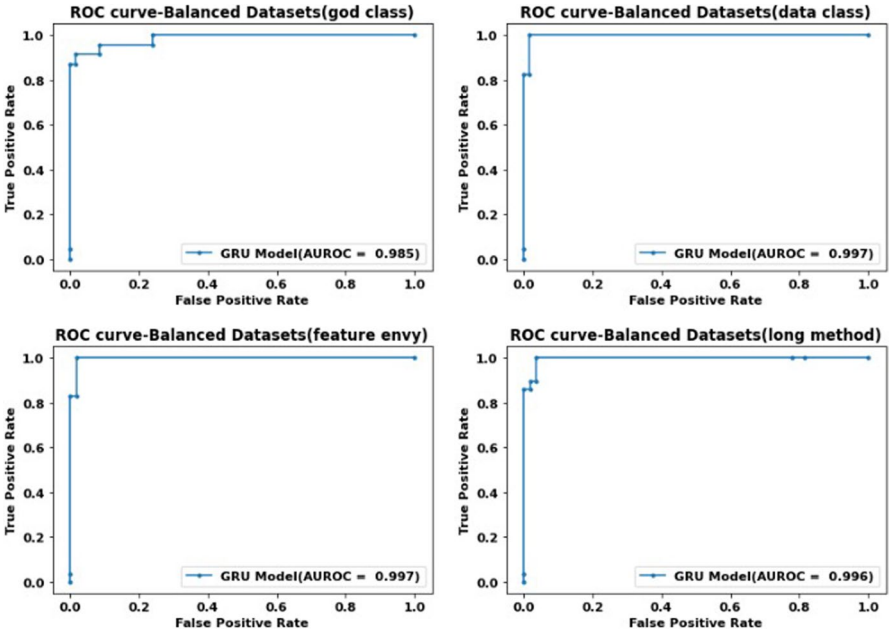


**Fig. 23** ROC curves for the balanced datasets—GRU Model-Tomek links

range from 0.96 to 1.00, the precision values range from 0.95 to 1.00, the recall value range from 0.98 to 1.00, the *F*-Measure values range from 0.97 to 1.00, the MCC values range from 0.92 to 1.00, the AUC values range from 0.96 to 1.00, the AUCPR values range from 0.93 to 1.00, and the MSE values range from 0.002 to 0.033 across all datasets.

Table 7 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, 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 8 shows the confusion matrix of our best model, the GRU model, based on the balanced datasets using the Tomek links technique. Regarding the God Class dataset, the actual nonsmelly cases are 58, our model predicted all of them correctly, and the actual smelly instances are 23, of which our model predicted 20 correctly. Regarding the Data Class dataset, the actual nonsmelly cases are 57, our model predicted 56 of them correctly, and the actual smelly instances are 23,

our model predicted all of them correctly. Regarding the Feature envy dataset, the actual nonsmelly cases are 52, our model predicted 51 of them correctly, and the actual smelly instances are 29, and our model predicted all of them correctly. Regarding the Long method dataset, the actual nonsmelly cases are 54, our model predicted 52 of them correctly, and the actual smelly instances are 29, our model predicted all of them correctly.

Figures 6, 7, 8 and 9 show the training and validation (accuracy and loss) of the proposed models on the original datasets.

Figures 6 and 7 show the training and validation accuracy of the models on the original datasets. The vertical axis presents the accuracy of the models, and the horizontal axis illustrates the number of epochs. Accuracy is the fraction of predictions that our models predicted right.

Figure 6 shows the accuracy values of the Bi-LSTM model. From Figure, the model learned 95% accuracy for God Class, 95% accuracy for Data Class, 95% accuracy for Feature envy, and 98% accuracy for Long method at the 100th epoch.

Figure 7 shows the accuracy values of the GRU model. From Figure, the model learned 93% accuracy for God Class, 96% accuracy for Data Class, 93% accuracy for Feature envy, and 98% accuracy for Long method at the 100th epoch.

Figures 8 and 9 show the training and validation loss of the models on the original datasets. The vertical axis presents the loss of the models, and the horizontal axis illustrates the number of epochs. The loss indicates how bad a model's prediction was.

Figure 8 shows the loss values of the Bi-LSTM model. From Figure, the model loss is 0.035 for God Class, 0.037 for Data Class, 0.044 for Feature envy, and 0.023 for the long method at the 100th epoch.

Figure 9 shows the loss values of the GRU model. From Figure, the model loss is 0.063 for God Class, 0.026 for Data Class, 0.065 for Feature envy, and 0.020 for the long method at the 100th epoch.

As shown in Figures, the accuracy of training and validation increases, and the loss decreases with increasing epochs. Regarding the high accuracy and low loss obtained by the proposed models, we note that both models are well-trained and validated. Additionally, we note that the models are approximately perfectly fitting, there are no overfitting or underfitting.

Figures 10, 11, 12, 13, 14, 15, 16 and 17 show the training and validation (accuracy and loss) of the proposed models on the balanced datasets.

Figures 10, 11, 12 and 13 show the training and validation accuracy of the models on the balanced datasets. The vertical axis presents the accuracy of the models, and the horizontal axis illustrates the number of epochs. Accuracy is the fraction of predictions that the models predicted right.

Figure 10 shows the accuracy values of the Bi-LSTM model with the Random Oversampling technique. From Figure, the model learned 96% accuracy for God Class, 99% accuracy for Data Class, 96% accuracy for Feature envy, and 100% accuracy for Long method at the 100th epoch.

Figure 11 shows the accuracy values of the Bi-LSTM model with the Tomek links technique. From Figure, the model learned 96% accuracy for God Class,

95% accuracy for Data Class, 98% accuracy for Feature envy, and 99% accuracy for Long method at the 100th epoch.

Figure 12 shows the accuracy values of the GRU model with the Random Oversampling technique. From Figure, the model learned 96% accuracy for God Class, 98% accuracy for Data Class, 97% accuracy for Feature envy, and 100% accuracy for Long method at the 100th epoch.

Figure 13 shows the accuracy values of the GRU model with the Tomek links technique. From Figure, the model learned 96% accuracy for God Class, 99% accuracy for Data Class, 99% accuracy for Feature envy, and 98% accuracy for Long method at the 100th epoch.

Figures 14, 15, 16 and 17 show the training and validation loss of the models on the balanced datasets. The vertical axis presents the loss of the models, and the horizontal axis illustrates the number of epochs. The loss indicates how bad a model's prediction was.

Figure 14 shows the loss values of the Bi-LSTM model with the Random Oversampling technique. From Figure, the model loss is 0.035 for God Class, 0.006 for Data Class, 0.037 for Feature envy, and 0.005 for the long method at the 100th epoch.

Figure 15 shows the loss values of the Bi-LSTM model with the Tomek links technique. From Figure, the model loss is 0.037 for God Class, 0.044 for Data Class, 0.020 for Feature envy, and 0.013 for the long method at the 100th epoch.

Figure 16 shows the loss values of the GRU model with the Random Oversampling technique. From Figure, the model loss is 0.033 for God Class, 0.023 for Data Class, 0.032 for Feature envy, and 0.002 for the long method at the 100th epoch.

Figure 17 shows the loss values of the GRU model with the Tomek links technique. From Figure, the model loss is 0.038 for God Class, 0.018 for Data Class, 0.021 for Feature envy, and 0.025 for the long method at the 100th epoch.

As shown in Figures, the accuracy of training and validation increases, and the loss decreases with increasing epochs. Regarding the high accuracy and low loss obtained by the proposed models, we note that both models are well-trained and validated. Additionally, we note that the models are approximately perfectly fitting, there is no overfitting or underfitting.

Figures 18 and 19 show the original datasets' ROC curves for both models. The vertical axis presents the actual positive rate of the models, and the horizontal axis illustrates the false-positive rate. The AUC is a sign of the performance of the model. The larger the AUC is, the better the model performance will be. Based on Figures, the values are very encouraging and indicate our proposed models' efficiency in detecting code smells.

Figure 18 shows the AUC values of the Bi-LSTM model. From Figure, the AUC values are 99% on God Class,99% on Data Class, 95% on Feature envy, and 99% on the Long method.

Figure 19 shows the AUC values of the GRU model. From Figure, the AUC values are 97% on God Class, 99% on Data Class, 89% on Feature envy, and 99% on the Long method.

Figures 20, 21, 22 and 23 show the ROC curves for both models on the balanced datasets. The vertical axis presents the actual positive rate of the models, and the

horizontal axis illustrates the false-positive rate. The AUC is a sign of the performance of the model. The larger the AUC is, the better the model performance will be.

Figure 20 shows the AUC values of the Bi-LSTM model with the Random Oversampling technique. From Figure, the AUC values are 98% on God Class, 100% on Data Class, 97% on Feature envy, and 100% on the Long method.

Figure 21 shows the AUC values of the Bi-LSTM model with the Tomek links technique. From Figure, the AUC values are 0.98% on God Class, 97% on Data Class, 99% on Feature envy, and 98% on the Long method.

Figure 22 shows the AUC values of the GRU model with the Random Oversampling technique. From Figure, the AUC values are 96% on God Class, 99% on Data Class, 97% on Feature envy, and 100% on the Long method.

Figure 23 shows the AUC values of the GRU model with the Tomek links technique. From Figure, the AUC values are 98% on God Class, 99% on Data Class, 99% on Feature envy, and 99% on the Long method.

## 6.1 Results of RQ1

### 6.1.1 RQ1 Do data balancing techniques improve DL models' accuracy in detecting code smells?

To answer RQ1, we test our proposed models on four types of code smells. The performance of the prediction models for the four code smells datasets is reported in Figs. 24, 25 and 26, and Tables 9 and 10.

Boxplots are very useful for describing the distribution of results and providing raw results for comparing different techniques. Therefore, we aggregated the achieved results to get a more accurate overview of the quality of the results using boxplots.

Figure 24 shows the Box plots for the original datasets' performance measures. For the Bi-LSTM model, the highest accuracy is 98% on the Long method dataset and the lowest accuracy is 95% on the God Class, Data Class, and Feature envy datasets, the highest precision is 100% on the Data Class dataset and the lowest precision is 93% on the Feature envy dataset, the highest recall is 96% on the

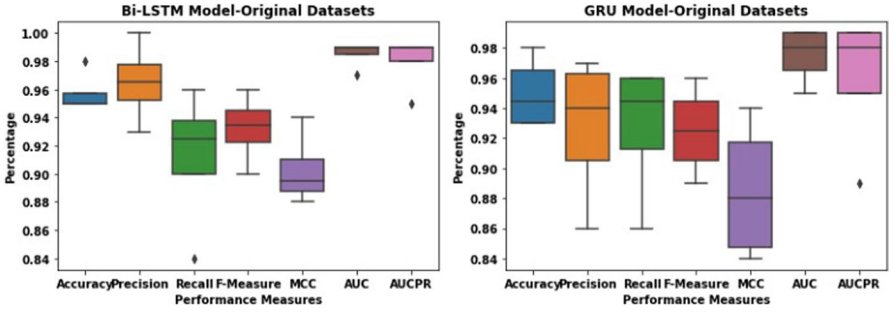


**Fig. 24** Boxplots representing performance measures obtained by models on the original datasets

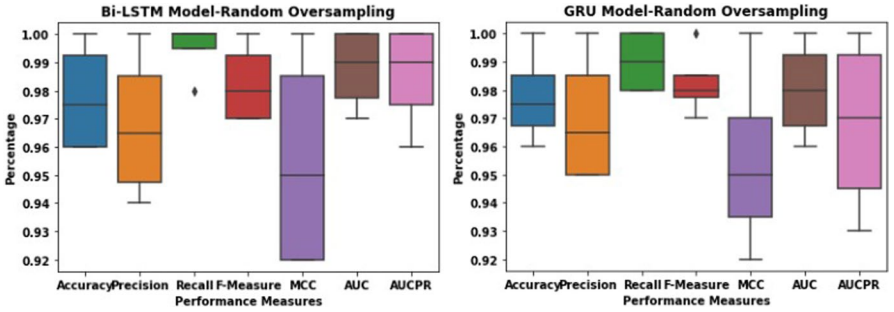


**Fig. 25** Boxplots representing performance measures obtained by models on the balanced datasets-random oversampling

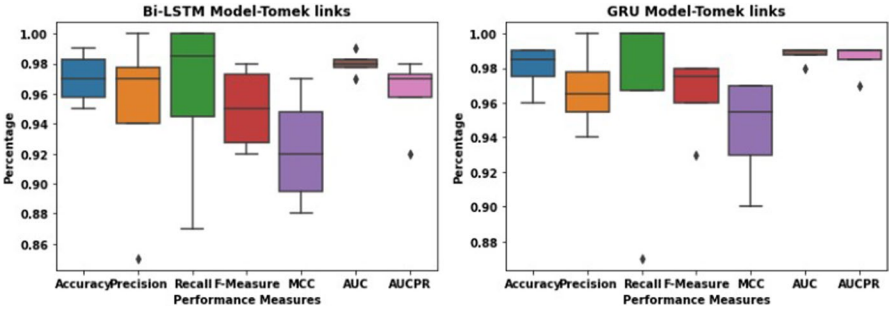


**Fig. 26** Boxplots representing performance measures obtained by models on the balanced datasets-Tomek links

**Table 9** Comparison of the proposed models in terms of accuracy using paired *t*-test- based on the original and balanced datasets (using random oversampling)

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

Long method dataset and the lowest recall is 83% on the Data Class dataset, the highest *f*-measure is 96% on the Long method dataset and the lowest *f*-measure is 90% on the Data Class dataset, the highest MCC is 94% on the Long method dataset and the lowest MCC is 88% on the Data Class dataset, the highest AUC is 99% on the God Class, Data Class and Long method datasets and the lowest AUC is 97% on the Feature envy dataset, the highest AUCPR is 99% on the God Class,

**Table 10** Comparison of the proposed models in terms of accuracy using paired *t*-test- based on the original and balanced datasets (using Tomek Links)

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

Data Class and Long method datasets and the lowest AUCPR is 95% on the Feature envy dataset.

In contrast, For the GRU model, the highest accuracy is 98% on the Long method dataset and the lowest accuracy is 93% on the God Class and Feature envy datasets, the highest precision is 97% on the God Class dataset and the lowest precision is 86% on the Feature envy dataset, the highest recall is 96% on the Data Class and Long method datasets and the lowest recall is 86% on the God Class dataset, the highest *f*-measure is 96% on the Long method dataset and the lowest *f*-measure is 89% on the Feature envy dataset, the highest MCC is 94% on the Long method dataset and the lowest MCC is 84% on the Feature envy dataset, the highest AUC is 99% on the Data Class and Long method datasets and the lowest AUC is 95% on the Feature envy dataset, the highest AUCPR is 99% on the Data Class and Long method datasets and the lowest AUCPR is 89% on the Feature envy dataset.

Figure 25 shows the Box plots for the performance measures on the balanced datasets using random oversampling. For the Bi-LSTM model with random oversampling, the highest accuracy is 100% on the Long method dataset and the lowest accuracy is 96% on the God Class and Feature envy datasets, the highest precision is 100% on the Long method dataset and the lowest precision is 94% on the Feature envy dataset, the highest recall is 100% on the Data Class, Feature envy and Long method datasets and the lowest recall is 98% on the God Class dataset, the highest *f*-measure is 100% on the Long method dataset and the lowest *f*-measure is 97% on the God Class and Feature envy datasets, the highest MCC is 100% on the Long method dataset and the lowest MCC is 92% on the God Class and Feature envy datasets, the highest AUC is 100% on the Data Class and Long method datasets and the lowest AUC is 97% on the Feature envy dataset, the highest AUCPR is 100% on the Data Class and Long method datasets and the lowest AUCPR is 96% on the Feature envy dataset.

In contrast, For the GRU model with random oversampling, the highest accuracy is 100% on the Long method dataset and the lowest accuracy is 96% on the God Class dataset, the highest precision is 100% on the Long method dataset and the lowest precision is 95% on the God Class and Feature envy datasets, the highest recall is 100% on the Feature envy and Long method datasets and the lowest recall is 98% on the God Class and Data Class datasets, the highest *f*-measure is

100% on the Long method dataset and the lowest *f*-measure is 97% on the God Class dataset, the highest MCC is 100% on the Long method dataset and the lowest MCC is 92% on the God Class dataset, the highest AUC is 100% on the Long method dataset and the lowest AUC is 96% on the God Class dataset, the highest AUCPR is 100% on the Long method dataset and the lowest AUCPR is 93% on the God Class dataset.

Figure 26 shows the Box plots for the performance measures on the balanced datasets using Tomek links. For the Bi-LSTM model with Tomek links, the highest accuracy is 99% on the Long method dataset and the lowest accuracy is 95% on the Data Class dataset, the highest precision is 100% on the God Class dataset and the lowest precision is 85% on the Data Class dataset, the highest recall is 100% on the Data Class and Long method datasets and the lowest recall is 87% on the God Class dataset, the highest *f*-measure is 98% on the Long method dataset and the lowest *f*-measure is 92% on the Data Class dataset, the highest MCC is 97% on the Long method dataset and the lowest MCC is 88% on the Data Class dataset, the highest AUC is 99% on the Feature envy dataset and the lowest AUC is 97% on the Data Class dataset, the highest AUCPR is 98% on the Feature envy dataset and the lowest AUCPR is 92% on the Data Class dataset.

In contrast, For the GRU model with Tomek links, the highest accuracy is 99% on the Data Class and Feature envy datasets and the lowest accuracy is 96% on the God Class dataset, the highest precision is 100% on the God Class dataset and the lowest precision is 94% on the Long method dataset, the highest recall is 100% on the Data Class, Feature envy and Long method datasets and the lowest recall is 87% on the God Class dataset, the highest *f*-measure is 98% on the Data Class and Feature envy datasets and the lowest *f*-measure is 93% on the God Class dataset, the highest MCC is 97% on the Data Class and Feature envy datasets and the lowest MCC is 90% on the God Class dataset, the highest AUC is 99% on the Data Class, Feature envy and Long method datasets and the lowest AUC is 98% on the God Class dataset, the highest AUCPR is 99% on the Data Class, Feature envy and Long method datasets and the lowest AUCPR is 97% on the God Class dataset.

Table 9 presents the statistical analysis results (paired *t*-test) of proposed models on the original and balanced datasets (using random oversampling) in terms of mean, Standard Deviation (STD), min, max, and *P* value. We notice that the mean values of the Bi-LSTM model are 0.95 on the original datasets and 0.97 on the balanced datasets. The mean values of the GRU model are 0.95 on the original datasets and 0.97 on the balanced datasets. The STD values of the Bi-LSTM model are 0.01 on the original datasets and 0.02 on the balanced datasets; while, the STD values of the GRU model are 0.02 on the original datasets and 0.01 on the balanced datasets. The Min values of the Bi-LSTM model are 0.95 on the original datasets and 0.96 on the balanced datasets; while, the Min values of the GRU model are 0.93 on the original datasets and 0.96 on the balanced datasets. The Max values of the Bi-LSTM model are 0.98 on the original datasets and 1.00 on the balanced datasets; while, the Max values of the GRU model are 0.98 on the original datasets and 1.00 on the balanced datasets. The *P* value of the Bi-LSTM model is 0.06 for the original and balanced datasets; while, the *P* value of the GRU model is 0.01 for the original and balanced datasets. Based on the *P* value of the GRU model on the original and

balanced data sets, we note that the *P* value is less than 0.05, indicating a difference between the results of the models on the original and balanced data sets.

Table 10 presents the statistical analysis results (paired *t*-test) of proposed models on the original and balanced datasets (using Tomek Links) in terms of mean, Standard Deviation (STD), min, max, and *P* value. We notice that the mean values of the Bi-LSTM model are 0.95 on the original datasets and 0.97 on the balanced datasets. The mean values of the GRU model are 0.95 on the original datasets and 0.98 on the balanced datasets. The STD values of the Bi-LSTM model are 0.01 on the original datasets and 0.01 on the balanced datasets; while, the STD values of the GRU model are 0.02 on the original datasets and 0.01 on the balanced datasets. The Min values of the Bi-LSTM model are 0.95 on the original datasets and 0.95 on the balanced datasets; while, the Min values of the GRU model are 0.93 on the original datasets and 0.96 on the balanced datasets. The Max values of the Bi-LSTM model are 0.98 on the original datasets and 0.99 on the balanced datasets; while, the Max values of the GRU model are 0.98 on the original datasets and 0.99 on the balanced datasets. The *P* value of the Bi-LSTM model is 0.14 for the original and balanced datasets; while, the *P* value of the GRU model is 0.09 for the original and balanced datasets. Based on the *P* value of both models on the original and balanced data sets, we note that the *P* value is greater than 0.05, indicating no difference between the results of the models on the original and balanced data sets.

## 6.2 Results of RQ2

### 6.2.1 RQ2 Which data balancing technique is the most effective at improving the accuracy of DL techniques?

To answer RQ2, the results of the models for the four code smells datasets are reported in Table 11; the best values are indicated in bold in Table.

**Table 11** Comparison of proposed models based on balanced datasets using the average of many performance measures

| Performance measures | Bi-LSTM Model | | GRU Model | |
|---|---|---|---|---|
| | Balanced datasets (using random oversampling) | Balanced datasets (using Tomek Links) | Balanced datasets (using random oversampling) | Balanced datasets (using Tomek links) |
| Accuracy | 0.97 | 0.97 | 0.97 | **0.98** |
| Precision | 0.96 | 0.94 | **0.97** | 0.96 |
| Recall | **0.99** | 0.96 | **0.99** | 0.96 |
| *F*-measure | **0.98** | 0.95 | **0.98** | 0.96 |
| MCC | **0.95** | 0.92 | **0.95** | 0.94 |
| AUC | **0.98** | **0.98** | **0.98** | **0.98** |
| AUCPR | **0.98** | 0.96 | 0.96 | **0.98** |
| MSE | **0.020** | 0.028 | 0.022 | 0.025 |

Bold values indicate the best values of the performance measures

### 6.3 Results of RQ3

#### 6.3.1 Does the proposed method outperform the state-of-the-art methods in detecting code smells?

To answer RQ3, the results presented by our models and previous studies' results are reported in Tables 12 and 13. We compared our results with the results obtained in previous studies based on two performance measures: accuracy and AUC. Tables 12 and 13 compare the values of performance measures obtained by our models and those of previous studies. Table 12 shows the results based on accuracy; Table 13 shows the results based on AUC. The best values are indicated in bold in Tables and "-" indicates that the approaches that did not use data balancing techniques or did not provide results for performance measures in a particular data set. According to Tables 12 and 13, some of the results in the previous studies are better than ours, but in most cases, our method outperforms the other state-of-the-art approaches and provides better predictive performance.

*The implication of the findings* The results should have implications for researchers and practitioners in code smell detection. They are interested in quantitatively understanding the effectiveness and efficiency of applying data balancing techniques with DL models in code smell detection. Furthermore, the formers are concerned about the qualitative perspective of the results. To summarize the main findings of our results and research questions, we provide implications related to effectiveness, efficiency, comparison, and relation with previous work, as follows:

Concerning RQ1, we observe from Figs. 24, 25 and 26 and Tables 9 and 10 that the proposed models perform better on the balanced datasets than on the original datasets. This indicates that data balancing techniques enhance the accuracy of DL models in detecting code smells.

Concerning RQ2, Table 11 shows that the results obtained on the balanced datasets using random oversampling are better than the results obtained using Tomek links. This suggests that random oversampling is the most effective technique for improving the accuracy of DL techniques in the detection of code smells.

Regarding RQ3, the comparison results of our proposed models with existing approaches using paired *t*-tests based on random oversampling and Tomek Links are presented in Tables 12 and 13. We observe that there is a difference between the results, indicating that our proposed models achieved better accuracy averages than the existing approaches.

*Threats to validity* This sub-section discusses our study's threats to validity and experiment limitations and how we mitigate them. It is essential to assess the threats to validity, such as construct, internal, external, and experiment limitations, particularly constraints on the search process and deviations from the standard practice.

Construct validity concerns the study's design and its possibility to reflect the actual goal of the research. To avoid threats in study design, we have applied a procedure of systematic literature review. To ensure that the research area is relevant to the study goal, we have cross-checked the research questions and adjusted them several times to address the business needs. Another threat is the construction of the DL models, for which we considered several aspects that could have influenced the

**Table 12** Comparison of the proposed models with other existing approaches based on the accuracy

| Approaches | | Datasets | | | |
|---|---|---|---|---|---|
| ML and DL models | Data balancing techniques | God class | Data class | Feature envy | Long method |
| Decision tree [2] | Random oversampling | 0.98 | **1.00** | **1.00** | 0.98 |
| *K*-nearest neighbors [2] | Random oversampling | 0.97 | 0.96 | 0.96 | 0.91 |
| Support vector machine [2] | Random oversampling | 0.96 | 0.97 | **1.00** | 0.96 |
| XGBoost [2] | Random oversampling | 0.96 | **1.00** | **1.00** | 0.98 |
| Multi-layer perceptron [2] | Random oversampling | 0.97 | 0.98 | 0.98 | 0.96 |
| Random forest [4] | – | 0.96 | 0.98 | 0.96 | 0.99 |
| Naive Bayes [4] | – | 0.97 | 0.97 | 0.91 | 0.97 |
| Decision tree [6] | – | – | – | 0.97 | – |
| Random forest [6] | – | – | 0.99 | – | 0.95 |
| DeleSmell [11] | – | – | 0.97 | – | 0.98 |
| CNN [13] | SMOTE | 0.96 | 0.98 | 0.98 | **1.00** |
| XGBoost [14] | SMOTE | **0.99** | – | – | – |
| Support vector machine [14] | SMOTE | 0.97 | – | – | – |
| *K*-nearest neighbors [14] | SMOTE | 0.97 | – | – | – |
| AdaBoost [19] | SMOTE | 0.98 | 0.99 | 0.98 | **1.00** |
| XGBoost [19] | SMOTE | 0.97 | **1.00** | 0.98 | **1.00** |
| CNN [19] | SMOTE | 0.98 | 0.99 | 0.99 | 0.99 |
| *K*-nearest neighbors [21] | – | 0.97 | 0.97 | 0.91 | 0.97 |
| Naive Bayes [21] | – | 0.96 | 0.84 | 0.92 | 0.95 |
| Multi-layer perceptron [21] | – | 0.97 | 0.97 | 0.95 | 0.96 |
| Decision tree [21] | – | 0.97 | 0.98 | 0.98 | 0.98 |
| Random forest [21] | – | 0.97 | 0.98 | 0.97 | 0.99 |
| Logistic regression [21] | – | 0.97 | 0.97 | 0.97 | 0.99 |
| Random forest [22] | – | 0.69 | 0.70 | 0.71 | 0.68 |
| Naive Bayes [22] | – | 0.82 | 0.75 | 0.83 | 0.81 |
| Support vector machine [22] | – | 0.74 | 0.83 | 0.83 | 0.81 |
| *K*-nearest neighbors [22] | – | 0.80 | 0.82 | 0.82 | 0.81 |
| Naive Bayes [39] | – | 0.96 | – | 0.91 | 0.97 |
| Multi-layer perceptron [39] | – | 0.97 | – | 0.92 | 0.99 |
| Decision tree [39] | – | 0.98 | – | 0.95 | 0.97 |
| Our Bi-LSTM model | Random oversampling | 0.96 | 0.99 | 0.96 | **1.00** |
| Our GRU model | Random oversampling | 0.96 | 0.98 | 0.97 | **1.00** |
| Our Bi-LSTM model | Tomek links | 0.96 | 0.95 | 0.98 | 0.99 |
| Our GRU model | Tomek links | 0.96 | 0.99 | 0.99 | 0.98 |

Bold values indicate the best values of the performance measures

**Table 13** Comparison of the proposed models with other existing approaches based on AUC

| Approaches | | Datasets | | | |
|---|---|---|---|---|---|
| ML and DL models | Data balancing techniques | God class | Data class | Feature envy | Long method |
| Decision tree [2] | Random oversampling | 0.98 | **1.00** | **1.00** | 0.98 |
| *K*-nearest neighbors [2] | Random oversampling | 0.97 | 0.98 | 0.97 | 0.93 |
| Support vector machine [2] | Random oversampling | **0.99** | 0.99 | **1.00** | 0.97 |
| XGBoost [2] | Random oversampling | 0.98 | **1.00** | **1.00** | 0.98 |
| Multi-layer perceptron [2] | Random oversampling | 0.98 | 0.99 | **1.00** | 0.98 |
| Deep learning [9] | – | – | – | 0.84 | 0.79 |
| XGBoost [14] | SMOTE | 0.96 | – | – | – |
| Support vector machine [14] | SMOTE | 0.88 | – | – | – |
| *K*-nearest neighbors [14] | SMOTE | 0.92 | – | – | – |
| Random forest [22] | – | 0.59 | 0.65 | 0.59 | 0.52 |
| Naive Bayes [22] | – | 0.88 | 0.85 | 0.86 | 0.86 |
| Support vector machine [22] | – | 0.65 | 0.88 | 0.82 | 0.66 |
| *K*-nearest neighbors [22] | – | 0.83 | 0.86 | 0.83 | 0.86 |
| Our Bi-LSTM model | Random oversampling | 0.98 | **1.00** | 0.97 | **1.00** |
| Our GRU model | Random oversampling | 0.96 | 0.99 | 0.97 | **1.00** |
| Our Bi-LSTM model | Tomek links | 0.98 | 0.97 | 0.99 | 0.98 |
| Our GRU model | Tomek links | 0.98 | 0.99 | 0.99 | 0.99 |

Bold values indicate the best values of the performance measures

study, i.e., data pre-processing, which features to consider, how to train the models, etc. However, the procedures followed in this respect are precise enough to ensure the study's validity.

Threats to internal validity are related to the correctness of the experiment's outcome or the study's process. The main threat to internal validity is datasets. The datasets used in our study are constructed from datasets published by Arcelli Fontana et al. [4]. The reference datasets are imbalanced datasets that show a lack of the real distribution of code smells and metrics. We manage this threat by modifying the original datasets to increase the realism of the data in terms of smells actual presence in the software system. The distribution of the dataset is modified by applying two data sampling methods.

External validity concerns the possibility of generalizing the study to a broader range of applications. We used four code smell datasets constructed from 74 open-source Java projects for our experimentation. However, we cannot declare that our results can be generalized to other coding languages, practitioners, and industrial

practices. Future replications of this study are necessary to confirm the generalizability of our findings.

The limitations of the experiments are summarized as follows. First, the number of code smells used in our experiments is limited to only two class-level and two method-level smells. Second, our findings may not be enough to generalize to all software in the industrial domain.

## 7 Conclusion

Code smells in the software systems indicate problems that can negatively affect software quality and make it hard to maintain, reuse, and expand. Therefore, detecting code smell is essential to enhance software quality, improve software maintainability, and reduce the risk of failure in the software system. This study presented a methodology based on DL algorithms combined with data balancing techniques 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 comprising 74 open-source software systems. We applied two data balancing techniques which are random oversampling and Tomek links to address the data imbalance problem, and then conducted a set of experiments using two DL models on Java projects in different application domains and evaluated smell detection accuracy using various performance measures. The results of the proposed models were compared with the state-of-the-art approaches in code smell detection. The experimental results show that, on average, the results obtained by both models on the balanced datasets (using random oversampling) were better than the results on the original datasets by 2%; while, the results obtained by both models on the balanced datasets (using Tomek links) better than the results on the original datasets by 3%, which indicates the best average accuracy was obtained on the balanced datasets and our models relying on random oversampling got the best performance. The experimental results showed that the data balancing techniques could improve the performance of DL models for code smell detection. The results also demonstrate that the proposed models significantly improve the average accuracy compared to the recent work on code smell detection. In future work, we will evaluate our models on various datasets and investigate their accuracy in detecting other code smell types. Additionally, we intend to combine more ML and DL algorithms with data balancing techniques to improve the accuracy of code smell detection.

**Acknowledgements** This article was carried out as part of the 2020-1.1.2-PIACI-KFI-2020- 00147 "OmegaSys - Lifetime planning and failure prediction decision support system for facility management services" project implemented with the support provided by the National Research, Development, and Innovation Fund of Hungary, financed under the 2020.1.1.2-PIACI KFI funding scheme.

**Author contributions** Nasraldeen Alnor Adam Khleel conducted the experiments, performed the data analyses, wrote the manuscript text, and prepared figures and tabular data. Károly Nehéz reviewed the experiments, the data analyses, manuscript text, and figures and tabular data. Both authors, 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. Open Access funding is provided by the 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

**Conflict of interest** The authors declared that they have no competing interests in 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 A, Jain S, Goel S, Dhiman G (2021) A review on machine-learning based code smell detection techniques in object-oriented software system(s). Recent Adv Electr Electr Eng (Former Recent Pat Electr Electr Eng) 14(3):290–303. https://doi.org/10.2174/2352096513999200922125839
2. Khleel NAA, Nehéz K (2023) Detection of code smells using machine learning techniques combined with data-balancing methods. Int J Adv Intell Inform 9(3):402–417. https://doi.org/10.26555/ijain.v9i3.981
3. Virmajoki J (2020) Detecting code smells using artificial intelligence: a prototype. LUT-yliopisto. https://urn.fi/URN:NBN:fi-fe2020092976199
4. Arcelli Fontana F, Mäntylä MV, Zanoni M, Marino A (2016) Comparing and experimenting machine learning techniques for code smell detection. Empir Softw Eng 21(3):1143–1191. https://doi.org/10.1007/s10664-015-9378-4
5. Guggulothu T, Moiz SA (2020) Code smell detection using multi-label classification approach. Softw Qual J 28(3):1063–1086. https://doi.org/10.1007/s11219-020-09498-y
6. Mhawish MY, Gupta M (2020) Predicting code smells and analysis of predictions: using machine learning techniques and software metrics. J Comput Sci Technol 35(6):1428–1445. https://doi.org/10.1007/s11390-020-0323-7
7. Pecorelli F, Di Nucci D, De Roover C, De Lucia A (2019) On the role of data balancing for machine learning-based code smell detection. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation. pp 19–24. https://doi.org/10.1145/3340482.3342744
8. Pecorelli F, Di Nucci D, De Roover C, De Lucia A (2020) A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. J Syst Softw 169:110693. https://doi.org/10.1016/j.jss.2020.110693
9. Liu H, Jin J, Xu Z, Zou Y, Bu Y, Zhang L (2019) Deep learning based code smell detection. IEEE Trans Softw Eng 47(9):1811–1837. https://doi.org/10.1109/TSE.2019.2936376

10. Sharma T, Efstathiou V, Louridas P, Spinellis D (2019) On the feasibility of transfer-learning code smells using deep learning. ACM Trans Softw Eng Methodol 1(1):1–34. https://doi.org/10.48550/arXiv.1904.03031
11. Zhang Y, Ge C, Hong S, Tian R, Dong C, Liu J (2022) DeleSmell: code smell detection based on deep learning and latent semantic analysis. Knowl-Based Syst 255(14):109737. https://doi.org/10.1016/j.knosys.2022.109737
12. Sharma T, Efstathiou V, Louridas P, Spinellis D (2021) Code smell detection by deep direct-learning and transfer-learning. J Syst Softw 176:110936. https://doi.org/10.1016/j.jss.2021.110936
13. Khleel NAA, Nehéz K (2022) Deep convolutional neural network model for bad code smells detection based on oversampling method. Indones J Electr Eng Comput Sci 26(3):1725–1735. https://doi.org/10.11591/ijeecs.v26.i3.pp1725-1735
14. Alkharabsheh K, Alawadi S, Kebande VR, Crespo Y, Fernández-Delgado M, Taboada JA (2022) A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: a study of God class. Inf Softw Technol 143:106736. https://doi.org/10.1016/j.infsof.2021.106736
15. Elhassan T, Aljurf M (2016) Classification of imbalance data using tomek link (t-link) combined with random under-sampling (rus) as a data reduction method. Glob J Technol Optim. https://doi.org/10.4172/2229-8711.S1:111
16. Li F, Zou K, Keung JW, Yu X, Feng S, Xiao Y (2023) On the relative value of imbalanced learning for code smell detection. Softw Pract Exp 53(10):1902–1927. https://doi.org/10.1002/spe.3235
17. Kaur J, Singh S (2016) Neural network based refactoring area identification in software system with object oriented metrics. Indian J Sci Technol 9(10):1–8. https://doi.org/10.17485/ijst/2016/v9i10/85110
18. Hadj-Kacem M, Bouassida N (2018) A hybrid approach to detect code smells using deep learning. In: Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering. pp 529–552
19. Dewangan S, Rao RS, Mishra A, Gupta M (2022) Code smell detection using ensemble machine learning algorithms. Appl Sci 12(20):10321. https://doi.org/10.3390/app122010321
20. Das AK, Yadav S, Dhal S (2019) Detecting code smells using deep learning. In: TENCON 2019–2019 IEEE Region 10 Conference (TENCON), Kochi. pp 2081–2086. https://doi.org/10.1109/TENCON.2019.8929628
21. Dewangan S, Rao RS, Mishra A, Gupta M (2021) A novel approach for code smell detection: an empirical study. IEEE Access 9:162869–162883. https://doi.org/10.1109/ACCESS.2021.3133810
22. Jain S, Saha A (2022) Rank-based univariate feature selection methods on machine learning classifiers for code smell detection. Evol Intel 15(1):609–638. https://doi.org/10.1007/s12065-020-00536-z
23. Pontillo V, Amoroso d'Aragona D, Pecorelli F, Di Nucci D, Ferrucci F, Palomba F (2024) Machine learning-based test smell detection. Empir Softw Eng 29(2):1–44. https://doi.org/10.1007/s10664-023-10436-2
24. Xu W, Zhang X. Multi-granularity code smell detection using deep learning method based on abstract syntax tree. https://doi.org/10.18293/SEKE2021-014
25. Fowler M (2018) Refactoring: improving the design of existing code. Addison-Wesley Professional
26. Oliveira D, Assunção WK, Souza L, Oizumi W, Garcia A, Fonseca B (2020) Applying machine learning to customized smell detection: a multi-project study. In: Proceedings of the 34th Brazilian Symposium on Software Engineering, Natal. pp 233–242. https://doi.org/10.1145/3422392.3422427
27. Rao RS, Dewangan S, Mishra A, Gupta M (2023) A study of dealing class imbalance problem with machine learning methods for code smell severity detection using PCA-based feature selection technique. Sci Rep 13(1):16245. https://doi.org/10.1038/s41598-023-43380-8
28. Rehef KK, Abbas AS (2024) Improving code smell detection using deep stacked autoencoder. Preprint at https://doi.org/10.20944/preprints202403.1848.v1
29. Draz MM, Farhan MS, Abdulkader SN, Gafar MG (2021) Code smell detection using whale optimization algorithm. Comput Mater Contin 68(2):1919–1935
30. Bansal A, Jayant U, Jain A (2022) Categorical analysis of code smell detection using machine learning algorithms. Intell Sustain Syst 333:703–712. https://doi.org/10.1007/978-981-16-6309-3_6
31. Verma Y (2021) Complete guide to bidirectional LSTM (with python codes). Analytics India Magazine Pvt Ltd. https://analyticsindiamag.com/complete-guide-to-bidirectional-lstm-with-python-codes/
32. Christopher O. Understanding LSTM networks—colah's blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs/ Accessed 24 Mar 2023

33. Tempero E, Anslow C, Dietrich J, Han T, Li J, Lumpe M, Melton H, Noble J (2010) The qualitas corpus: a curated collection of java code for empirical studies. In: 2010 Asia Pacific Software Engineering Conference, Sydney. pp 336–345. https://doi.org/10.1109/APSEC.2010.46
34. Medeiros N, Ivaki N, Costa P, Vieira M (2020) Vulnerable code detection using software metrics and machine learning. IEEE Access 8:219174–219198. https://doi.org/10.1109/ACCESS.2020.3041181
35. Sultana KZ, Anu V, Chong TY (2021) Using software metrics for predicting vulnerable classes and methods in Java projects: a machine learning approach. J Softwe: Evol Process 33(3):1–20. https://doi.org/10.1002/smr.2303
36. Mehboob B, Chong CY, Lee SP, Lim JMY (2021) Reusability affecting factors and software metrics for reusability: a systematic literature review. Softw Pract Exp 51(6):1416–1458. https://doi.org/10.1002/spe.2961
37. Di Nucci D, Palomba F, Tamburri DA, Serebrenik A, De Lucia A (2018) Detecting code smells using machine learning techniques: are we there yet?. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (Saner), Campobasso. pp 612–621. https://doi.org/10.1109/SANER.2018.8330266
38. Zhang Y, Ge C, Liu H, Zheng K (2024) Code smell detection based on supervised learning models: a survey. Neurocomputing 565(14):127014. https://doi.org/10.1016/j.neucom.2023.127014
39. Cruz D, Santana A, Figueiredo E (2020) Detecting bad smells with machine learning algorithms: an empirical study. In: Proceedings of the 3rd International Conference on Technical Debt, Seoul. pp 31–40. https://doi.org/10.1145/3387906.3388618
40. Jain S, Saha A (2021) Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection. Sci Comput Program 212:102713. https://doi.org/10.1016/j.scico.2021.102713



## Authors and Affiliations

**Nasraldeen Alnor Adam Khleel[1] · Károly Nehéz[1]**

✉ Nasraldeen Alnor Adam Khleel
nasr.alnor@uni-miskolc.hu

Károly Nehéz
aitnehez@uni-miskolc.hu

[1] Department of Information Engineering, University of Miskolc, Miskolc 3515, Hungary