Optimizing LSTM for Code Smell Detection: The Role of Data Balancing

Nasraldeen Alnor Adam Khleel, and Károly Nehéz

Abstract—**Code smells are specific patterns or characteristics in software code that indicate potential design or implementation problems. Identifying code smells has gained significant attention in software engineering. It is essential to address code smells to maintain high-quality software systems. Machine learning (ML) models, such as Long Short-Term Memory (LSTM), have been to detect code smells automatically based on source code features. However, the imbalanced distribution of code smells within software projects poses a challenge to the accuracy of these models. This study explores the role of data balancing methods in optimizing the accuracy of the LSTM model for code smell detection. We investigate different techniques for addressing the class imbalance problem, including random oversampling and synthetic minority oversampling techniques (SMOTE). We evaluate the performance of the LSTM model with and without data balancing methods using accuracy, precision, recall, f-measure, Matthew's correlation coefficient (MCC), and the area under a receiver operating characteristic curve (AUC). Our experimental results, conducted on four code smell datasets (God class, data class, feature envy, and long method) extracted from 74 open-source systems, demonstrate the effectiveness of data balancing methods in improving the accuracy of the LSTM model for code smell detection. The results indicate that the use of data balancing methods had a positive effect on the predictive accuracy of the LSTM model. In addition, we compared our proposed method with state-of-the-art code smell detection approaches. The findings from the comparison indicate that our proposed method performs notably better than existing state-of-the-art approaches across the majority of datasets.**

Index Terms—**Software engineering, artificial intelligence, code smells, LSTM, software metrics, class imbalance, data balancing methods.**

I. Introduction

In software development, code smells are indicators of potential problems or design flaws that can degrade the quality of software systems [1, 2, 3]. Identifying and rectifying code smells ensures maintainable, efficient, and robust software applications [4, 5, 6]. In Table 1, we outline the four types of code smells examined in our study. Detection methods for code smells vary, including manual, automatic, and metrics-based approaches [7, 8, 9]. Nonetheless, the majority of these techniques adopt a heuristic two-step method. Initially, they compute metrics and subsequently utilize threshold values to distinguish between smelly and non-smelly classes. Differences among these approaches stem from the algorithms used, subjective interpretations, absence of consensus among detectors, and reliance on thresholds [1, 10]. Recently, researchers have embraced ML techniques to overcome the constraints in code smell detection. Their goal is to bypass the use of thresholds and decrease the occurrence of false positives in detection tools [6, 10]. Among these ML techniques, LSTM models have gained significant attention. LSTM is a type of recurrent neural network architecture that is designed to capture long-term dependencies and relationships in sequential data [11]. By leveraging software metrics as input features, LSTM models can learn patterns and relationships to identify code smells effectively [12].

However, one critical challenge in building accurate ML models for code smell detection lies in the imbalance of data. Data imbalance in classification models represents those situations where the number of examples of one class is much smaller than another [6, 8, 12, 13]. This imbalance can negatively impact the performance of ML models, leading to biased predictions and lower accuracy [1, 9].

Consequently, addressing the data imbalance problem becomes crucial for achieving reliable and robust code smell detection [10]. The dataset utilized for code smell detection in this research exhibits a significant imbalance. Consequently, the goal of this study is to employ data balancing methods like random oversampling and SMOTE to tackle the class imbalance issue and assess their effect on the performance of the LSTM model in code smell detection. In brief, our study aims to achieve the following objectives and make the following key contributions:

(i) This study identifies the data imbalance problem as a major challenge for machine learning techniques in detecting code smells.

(ii) To address the data imbalance problem and investigate the impact of data balancing methods in improving code smell detection, we propose a new method that combines the LSTM network with two data balancing methods (Random Oversampling and SMOTE).

(iii) We demonstrate that balancing the dataset can greatly enhance the performance of the LSTM model in code smell detection. Additionally, our approach surpasses existing state-of-the-art approaches for code smell detection.

The paper follows this structure: Section 2 introduces the LSTM network. Section 3 details the research method. Section 4 presents the results and discussions. The conclusion is provided in the final section, Section 5.

Nasraldeen Alnor Adam Khleel, and Károly Nehéz are with the University of Miskolc, Miskolc, Hungary
(E-mail: nasr.alnor@uni-miskolc.hu, aitnehez@uni-miskolc.hu)

DOI: 10.36244/ICJ.2024.3.5

II. Related work

Recently, there has been an increased focus on researching code smell detection, with numerous scientific studies utilizing ML models for this purpose. For example, Fabiano Pecorelli et al. [6] investigated five data balancing methods that were able to mitigate data unbalancing issues to understand their impact on ML algorithms for code smell detection. The experiment was performed based on five code smell datasets extracted from 13 open-source systems. The experimental results showed that the ML models relying on SOMTE realize the best performance. Hadj-Kacem and Nadia. [7] proposed a hybrid approach based on deep Autoencoder and artificial neural network algorithms to detect code smells. The approach was evaluated based on four code smells extracted from 74 open-source systems. The experiment results showed that the values of recall and precision measurements have demonstrated high accuracy results. Francesca Arcelli Fontana et al. [8] presented a method using different ML algorithms to detect four code smells based on 74 software systems. The results showed that all algorithms performed well, but unbalanced data caused some models' performances. Chhabr and Nanda. [9] proposed a new approach called the SMOTE-Stacked hybrid model (SSHM) for the severity classification of four code smells (God class, Data class, Feature envy, and Long method). The SMOTE method was used to address the problem of class imbalance.

TABLE I
LISTS THE FOUR SPECIFIC CODE SMELLS THAT WE HAVE INVESTIGATED [8]

Code smells	Description	Affected entity
God _Class	A god class refers to classes that have numerous members and execute various behaviors.	Class
Data _Class	A data Class is a class that has only data without functions or any behaviors and does not process this data.	Class
Long _Method	The long method refers to the method that is too long and increases the system's compatibility.	Method
Feature _Envy	Feature envy describes a method that shows more interest in the properties of other classes than its own.	Method

The Experimental results demonstrated that the proposed approach surpassed other literature studies with peak accuracy improvement to 97–99% from 76 to 92% for various code smells. Khleel and Nehéz. [1, 10, 12] presented various classical and advanced machine learning algorithms with data balancing methods to detect code smells based on a set of Java projects. The authors examined four datasets related to code smells (God class, data class, feature envy, and long method) and compared the results using various performance metrics. The experiments demonstrated that the models proposed, along with data balancing methods, exhibited improved performance in detecting code smells. Tushar.Sharma et al. [11] proposed a new method for code smell detection using convolution neural networks and recurrent neural networks. The experiments were conducted based on C# sample codes. The experiment results showed that it is feasible to detect smells using deep learning methods, and transfer-learning is possible to detect code smells with a performance like that of direct learning. Mohammad Y. Mhawish and Manjari Gupta [15] 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.

After reviewing previous studies in code smell detection, we noticed that the studies that dealt with and addressed the issue of class imbalance point out that the data balancing methods have an essential role in improving the accuracy of code smell detection [1, 6, 9, 10, 12]. So, the primary point from the recent studies is that ML combined with data balancing methods can improve and increase prediction accuracy. Therefore, our study focuses on addressing the class imbalance problem using random oversampling and SMOTE methods.

III. LSTM Network

Long Short-Term Memory (LSTM) networks, a specialized variant of recurrent neural network architecture, are engineered to detect intricate patterns within sequential data. The purpose of introducing LSTM networks was to resolve or avoid the problem of long-term dependencies, which regular recurrent neural networks are susceptible to due to an unstable gradient when connecting previous information to new information [11]. 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[14]. The interacting layers of the repeating module in an LSTM Network are depicted in Figure 1.

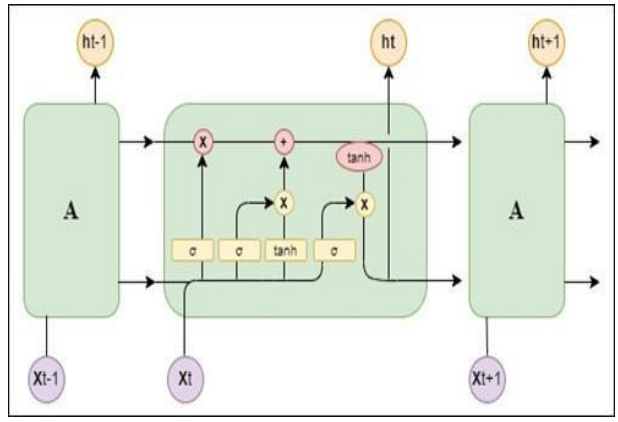

Fig. 1. Shows the interacting layers of the repeating module in an LSTM network.

IV. METHOD

Our study proposes a method for training and testing the code smell detection model, which utilizes the LSTM model with data balancing methods. Figure 2 illustrates an overview of the proposed method. The following sections describe the steps taken in this study, which encompass dataset description, data pre-processing and feature selection, class imbalance and data balancing methods, and model building and evaluation.

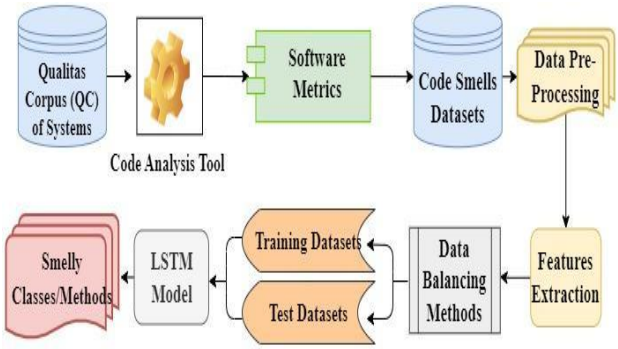

Fig. 2. Overview of the proposed method for code smell detection.

A. Dataset Description

For conducting the analysis and experiments, we implemented our method using datasets introduced by Arcelli Fontana et al. [8]. These datasets comprise 74 open-source systems of different sizes and domains gathered from the Qualitas Corpus (QC), encompassing 6,785,568 lines of code, 3,420 packages, and 51,826 classes [4]. These datasets were chosen because the systems must accurately compute metric values. Additionally, they are freely accessible, allowing researchers to iterate, compare, and evaluate their studies. Software metrics serve as widely utilized indicators of software quality, and numerous studies have demonstrated their effectiveness in estimating the presence of vulnerabilities or defects in code [13]. Software metrics help identify patterns and indicators associated with software code smells [14]. These metrics fall into two categories: static code metrics, which are directly derived from source code, and process metrics, which are obtained from the source code management system by analyzing historical changes in the codebase. 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., size, complexity, cohesion, size, coupling, encapsulation, and Inheritance [8].

B. Data Pre-processing and Features Selection

Pre-processing the gathered data is a crucial step before building the model. Ensuring high data quality is essential for creating an effective model. 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 [10, 13, 14]. Data pre-processing involves employing a range of methods to improve data quality before building a model. These methods include tasks like removing noise and undesirable outliers from the dataset, addressing missing values, converting feature types, and more [10, 11, 15]. Feature Selection (FS) is a crucial step in selecting the most discriminative features from the list of features using appropriate FS methods [10, 13, 16]. FS endeavors to select the most relevant features for the target class from high-dimensional features while eliminating redundant and uncorrelated ones. Feature extraction facilitates the conversion of pre-processed data into a form that the classification engine can use [3, 11, 17].

C. Class imbalance and data balancing methods

Class imbalance is one of the big challenges facing machine learning models [10, 13]. In classification models, class imbalance occurs when one class has significantly fewer examples than another. Hence, the class imbalance problem makes classification models not effectively predict minority modules [1, 18]. Numerous methods have been created to tackle the challenge of class imbalance, encompassing approaches like cost-sensitive learning, algorithmic adjustments, ensemble techniques, feature selection strategies, data sampling methodologies, and more. The most common among these methods are data sampling methods. These methods typically modify the initial distribution of both the majority and minority classes in the training dataset to achieve a more balanced class distribution.

Random oversampling and SMOTE are widely used data sampling techniques aimed at addressing class imbalance by augmenting the representation of the minority class [9, 14, 18]. Random oversampling involves duplicating instances from the minority class until a desired balance between classes is achieved [1]. Unlike random oversampling, which duplicates existing instances, SMOTE generates synthetic samples for the minority class based on the characteristics of its existing instances [10, 14]. The original datasets were composed of 561 smelly instances and 1119 non-smelly 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 non-smelly instances is 280), for Data Class (the number of smelly instances is 140 and the number of non-smelly 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 non-smelly instances is 280), for Long Method (the number of smelly instances is 141 and the number of non-smelly instances is 279).To address the problem of class imbalance and increase the realism of the data, we changed the distribution of instances using two algorithms: Random Oversampling and SMOTE. After balancing the datasets using these algorithms, each type of code smell had an equal number of instances. So, for God Class, there were 280 smelly instances and 280 non-smelly instances. The same goes for Data Class and Feature Envy. For the Long Method, there were 279 smelly instances and 279 non-smelly instances.

Figure 3 shows the distribution of learning instances over original and balanced datasets.

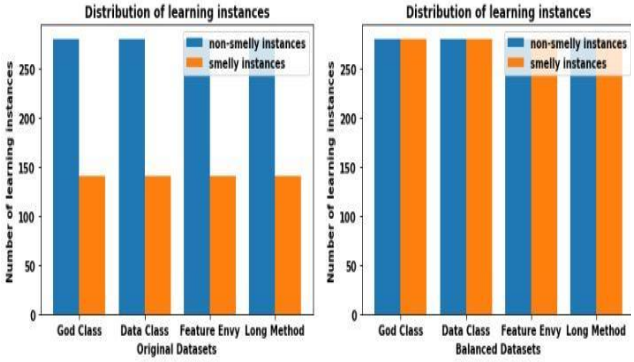

Fig. 3. Distribution of learning instances over original and balanced datasets.

D. Model Building and Evaluation

The model proposed in this study was built using Python programming language based on Keras, a high-level API that is based on TensorFlow. The training datasets constituted 80% of the datasets with randomly selected features, while the validation and test datasets constituted 20%. The model was developed using several parameters such as ReLU and sigmoid as activation functions, Adam as an optimizer, learning rate(0.01), mean squared error as loss function, batch size (64), and number of epochs (100).

The performance of the proposed model is assessed by utilizing a range of performance measures derived from the confusion matrix, MCC, and AUC. MCC is a performance metric that quantifies the difference between a model's predicted and actual values [13, 14]. AUC is a graph that shows how well a classification model performs at different threshold levels, plotting the true positive rate (TPR) against the false-positive rate (FPR) [10, 13, 14]. A confusion matrix is a tabular representation that summarizes the results of the testing algorithm, and it is commonly used to evaluate the performance of a model. Accuracy, precision, recall, and f-measure are the commonly used performance measurement parameters based on the confusion matrix. These parameters report the number of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions, which are presented in Table 2 [5, 10, 20].

TABLE II
CONFUSION MATRIX

Predicted	Actual	
	Class X	Class Y
Class X	TN	FP
Class Y	FN	TP

$$Accuracy = \frac{(TP+TN)}{(TP+FP+FN+TN)} \quad (1)$$

$$Precision = \frac{TP}{(TP+FP)} \quad (2)$$

$$Recall = \frac{TP}{(TP+FN)} \quad (3)$$

$$F - Measure = \frac{(2 * Recall * Precision)}{(Recall + Precision)} \quad (4)$$

$$MCC = (TP * TN - FP * FN) / \sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)} \quad (5)$$

$$AUC = \frac{\sum_{ins_i \in Positive\ Class} rank(ins_i) - \frac{M(M+1)}{2}}{M \cdot N} \quad (6)$$

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

V. RESULTS AND DISCUSSION

The experimental setup utilized the Python programming language, and the training and validation datasets were sourced from the same project. To ensure the credibility of the performance assessment, the proposed model underwent training and testing on extensive datasets containing over 6,785,568 lines of source code. Tables 3 and 4 and Figures 4 to 7 below show the results.

TABLE III
EVALUATION RESULTS OF THE LSTM MODEL ON THE ORIGINAL DATASETS

Datasets	Performance Measures					
	Accuracy	Precision	Recall	F-measure	MCC	AUC
God Class	0.93	0.89	0.89	0.89	0.83	0.94
Data Class	0.94	0.85	1.00	0.92	0.87	0.97
Feature Envy	0.95	0.90	0.96	0.93	0.89	0.99
Long Method	0.86	0.79	0.79	0.79	0.68	0.90
Average	**0.92**	**0.85**	**0.91**	**0.88**	**0.81**	**0.95**

Table 3 presents the results of the LSTM model based on the original datasets in terms of accuracy, precision, recall, f-measure, MCC, and AUC. We notice that the highest accuracy was achieved on Feature Envy, which is 95%, and the lowest accuracy was achieved on Long Method, which is 86%. The highest precision was achieved on Feature Envy, which is 90%, and the lowest precision was achieved on Long Method, which is 79%. The highest recall was achieved on Data Class, which is 100%, and the lowest recall was achieved on Long Method, which is 79%. The highest f-measure was achieved on Feature Envy, which is 93%, and the lowest f-measure was achieved on Long Method, which is 79%. The highest MCC was achieved on Feature Envy, which is 89%, and the lowest MCC was achieved on Long Method, which is 68%. The highest AUC was achieved on Feature Envy, which is 99%, and the lowest AUC was achieved on Long Method, which was 90%.

TABLE IV
EVALUATION RESULTS OF THE LSTM MODEL ON THE BALANCED DATASETS

Random Oversampling						
Datasets	**Performance Measures**					
	Accuracy	**Precision**	**Recall**	**F-measure**	**MCC**	**AUC**
God Class	0.97	0.95	1.00	0.98	0.94	0.98
Data Class	0.99	0.98	1.00	0.99	0.98	0.99
Feature Envy	0.96	0.92	1.00	0.96	0.91	0.97
Long Method	0.97	0.98	0.96	0.97	0.94	0.99
Average	**0.97**	**0.95**	**0.99**	**0.97**	**0.94**	**0.98**
SMOTE						
Datasets	**Performance Measures**					
	Accuracy	**Precision**	**Recall**	**F-measure**	**MCC**	**AUC**
God Class	0.97	0.95	1.00	0.98	0.94	0.98
Data Class	0.99	1.00	0.98	0.99	0.98	1.00
Feature Envy	0.96	0.95	0.97	0.96	0.90	0.99
Long Method	0.98	0.96	1.00	0.98	0.96	1.00
Average	**0.97**	**0.96**	**0.98**	**0.97**	**0.94**	**0.99**

Table 4 presents the results of the LSTM model based on the balanced datasets (using Random Oversampling and SMOTE) in terms of accuracy, precision, recall, f-measure, MCC, and AUC.

Regarding Random Oversampling: We notice that the highest accuracy was achieved on Data Class, which is 99%, and the lowest accuracy was achieved on Feature Envy, which is 96%. The highest precision was achieved on Data Class and Long Method, which is 98%, and the lowest precision was achieved on Feature Envy, which is 92%. The highest recall was achieved on God Class, Data Class, and Feature Envy, which is 100%, and the lowest recall was achieved on Long Method, which is 96%. The highest f-measure was achieved on Data Class, which is 99%, and the lowest f-measure was achieved on Feature Envy, which is 96%. The highest MCC was achieved on Data Class, which is 98%, and the lowest MCC was achieved on Feature Envy, which is 91%. The highest AUC was achieved on Data Class and Long Method, which is 99%, and the lowest AUC was achieved on Feature Envy, which was 97%.

Regarding SMOTE: We notice that the highest accuracy was achieved on Data Class, which is 99%, and the lowest accuracy was achieved on Feature Envy, which is 96%. The highest precision was achieved on Data Class, which is 100%, and the lowest precision was achieved on God Class and Feature Envy, which is 95%. The highest recall was achieved on God Class and Long Method, which is 100%, and the lowest recall was achieved on Feature Envy, which is 97%. The highest f-measure was achieved on Data Class, which is 99%, and the lowest f-measure was achieved on Feature Envy, which is 96%. The highest MCC was achieved on Data Class, which is 98%, and the lowest MCC was achieved on Feature Envy, which is 90%. The highest AUC was achieved on Data Class and Long Method, which is 100%, and the lowest AUC was achieved on God Class, which was 98%.

Figures 4 and 5 show the training and validation accuracy of the model on the balanced datasets. The vertical axis presents the model's accuracy, and the horizontal axis illustrates the number of epochs. Accuracy is the fraction of predictions that our model predicted right.

Figure 4 shows the accuracy values of the LSTM model on the balanced datasets (using Random Oversampling). From the Figure, the model learned 97% accuracy for God Class, 99% for Data Class, 96% for Feature Envy, and 97% for the Long method at the 100th epoch.

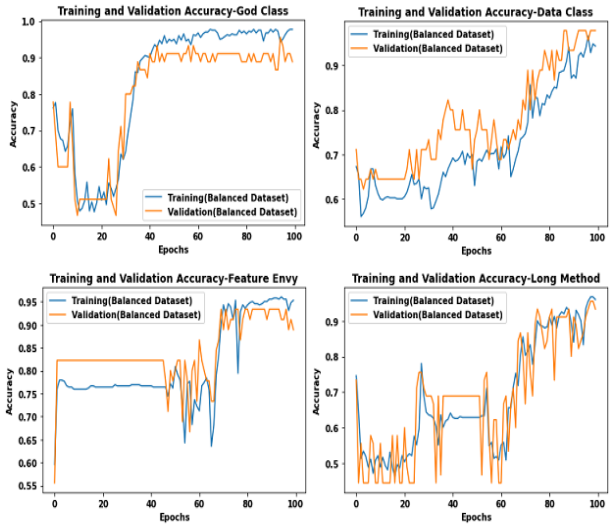

Fig. 4. Training and validation accuracy of LSTM model on the balanced datasets-random oversampling.

Figure 5 shows the accuracy values of the LSTM model on the balanced datasets (using SMOTE). From the Figure, the model learned 97% accuracy for God Class, 99% accuracy for Data Class, 96% accuracy for Feature Envy, and 98% accuracy for Long method at the 100th epoch.

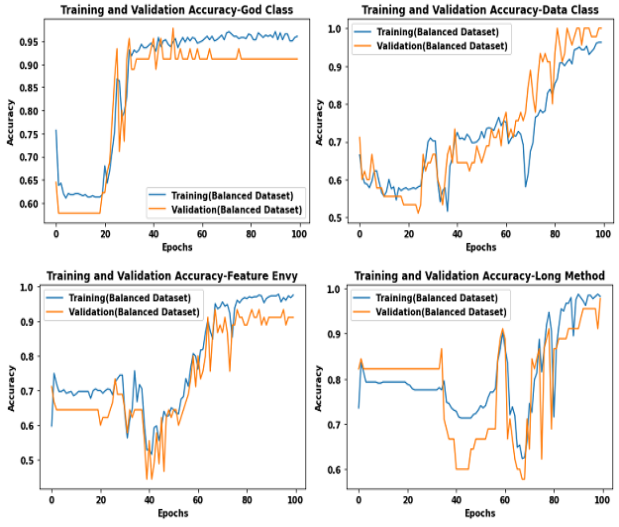

Fig. 5. Training and validation accuracy of LSTM model on the balanced datasets-SMOTE.

Figures 6 and 7 show the training and validation loss of the model on the balanced datasets. The vertical axis presents the loss of the model, and the horizontal axis illustrates the number of epochs. The loss indicates how wrong a model prediction was.

Figure 6 shows the loss values of the LSTM model on the balanced datasets (using Random Oversampling). From the Figure, the model loss is 0.028 for God Class, 0.013 for Data Class, 0.043 for Feature Envy, and 0.025 for the long method at the 100th epoch.

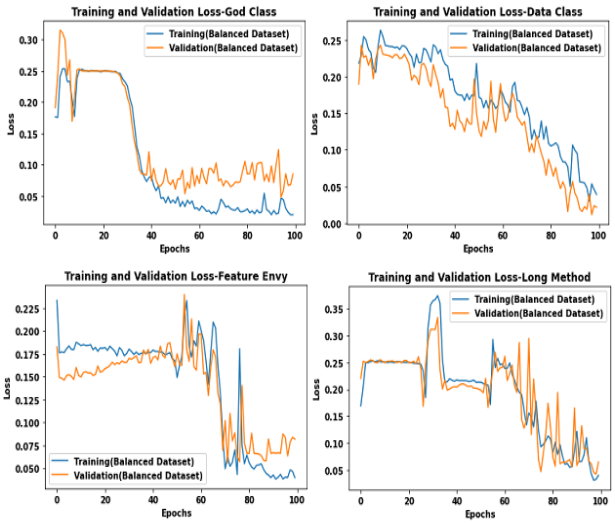

Fig. 6. Training and validation loss of LSTM model on the balanced datasets-random oversampling.

Figure 7 shows the loss values of the LSTM model on the balanced datasets (using SMOTE). From the Figure, the model loss is 0.034 for God Class, 0.009 for Data Class, 0.040 for Feature Envy, and 0.017 for the long method at the 100th epoch.

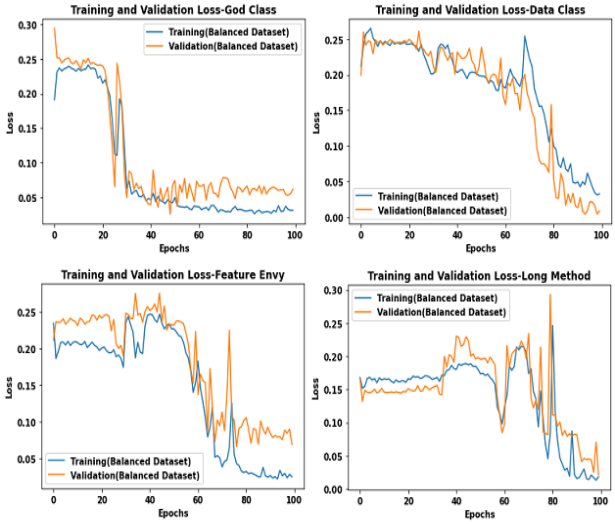

Fig. 7. Training and validation loss of LSTM model on the balanced datasets-SMOTE.

As illustrated in the figures, both training and validation accuracies improve while loss decreases as epochs progress. The high accuracy and low loss achieved by the proposed LSTM model indicate effective training and validation. Furthermore, it's worth mentioning that the model demonstrates almost ideal fitting, with no signs of overfitting or underfitting.

We compared our method results with the results obtained in previous studies based on the accuracy measure. Table 5 compares the values of accuracy obtained by our models and those of previous studies. The optimal values are highlighted in bold within the Table, while "-" indicates approaches that didn't provide results for a specific dataset. From Table 5, while certain results from past studies outshine ours, our method generally surpasses other state-of-the-art approaches, offering superior predictive performance.

TABLE V
COMPARISON OF THE PROPOSED MODELS WITH OTHER EXISTING APPROACHES BASED ON THE ACCURACY

Approaches	Datasets				Averages
	God class	Data class	Feature envy	Long method	
Decision Tree [1]	**0.98**	**1.00**	**1.00**	0.98	**0.99**
K-Nearest Neighbors [1]	0.97	0.96	0.96	0.91	0.95
Support Vector Machine [1]	0.96	0.97	**1.00**	0.96	0.97
XGBoost [1]	0.96	**1.00**	**1.00**	0.98	0.98
Multi-Layer Perceptron [1]	0.97	0.98	0.98	0.96	0.97
Random Forest (3)	0.69	0.70	0.71	0.68	0.69
Naive Bayes (3)	0.82	0.75	0.83	0.81	0.80
Support Vector Machine (3)	0.74	0.83	0.83	0.81	0.80
K-nearest neighbours (3)	0.80	0.82	0.82	0.81	0.81
K-nearest neighbours (5)	0.97	0.97	0.91	0.97	0.95
Naive Bayes (5)	0.96	0.84	0.92	0.95	0.91
Multi-layer Perceptron (5)	0.97	0.97	0.95	0.96	0.96
Decision Tree (5)	0.97	0.98	0.98	0.98	0.97
Random Forest (5)	0.97	0.98	0.97	**0.99**	0.97
Logistic Regression (5)	0.97	0.97	0.97	**0.99**	0.97
Random Forest (8)	0.96	0.98	0.96	**0.99**	0.97
Naive Bayes (8)	0.97	0.97	0.91	0.97	0.95
Decision Tree (15)	-	-	0.97	-	0.97
Random Forest (15)	-	0.99	-	0.95	0.97
Our LSTM model_Balanced Datasets (Random Oversampling)	0.97	0.99	0.96	0.97	0.97
Our LSTM model_Balanced Datasets (SMOTE)	0.97	0.99	0.96	0.98	0.97

VI. CONCLUSION

This study investigated the role and effectiveness of data balancing methods in optimizing the accuracy of the LSTM model for code smell detection. We introduced a novel method that combines the LSTM model with data balancing methods to improve upon current state-of-the-art methods for code smell detection. We addressed the challenge posed by imbalanced distributions of code smells within software

projects and investigated various data balancing methods, including random oversampling and SMOTE. To assess the efficiency of our proposed method, we conducted a series of experiments using four datasets on code smells. The average accuracy of our proposed LSTM model on both the original and balanced datasets (utilizing random oversampling and SMOTE) was 92%, 97%, and 97%, respectively.

The findings indicate that employing balanced datasets with the proposed model enhances the average accuracy by 5% in comparison to using the original datasets. Our experimental evaluation showcased the substantial improvement in the accuracy of the LSTM model for code smell detection through the implementation of data balancing methods. Moreover, our proposed method outperforms existing state-of-the-art approaches in code smell detection. We observe incorporating appropriate data balancing methods not only enhances the model's ability to detect code smells accurately but also mitigates the bias towards the majority class, resulting in a more balanced performance across different classes of code smells. This research has practical implications for software developers and researchers. It highlights the significance of considering data balancing methods when applying the LSTM model for code smell detection. By employing these methods, developers can enhance their ability to identify and address code quality issues, improving software maintainability.

ACKNOWLEDGMENT

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.

REFERENCES

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

[2] A. Al-Shaaby, H. Aljamaan and M. Alshayeb, "Bad smell detection using machine learning techniques: a systematic literature review", *Arabian Journal for Science and Engineering*, Vol. 45, No. 4, pp. 2341–2369, 2020. DOI: 10.1007/s13369-019-04311-w

[3] S. Jain and A. Saha, "Rank-based univariate feature selection methods on machine learning classifiers for code smell detection", *Evolutionary Intelligence*, Vol. 15, No. 1, pp. 609–638, 2022. DOI: 10.1007/s12065-020-00536-z

[4] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies", In *2010 Asia pacific software engineering conference*, Sydney, NSW, Australia, pp. 336–345, IEEE, 2010. DOI: 10.1109/APSEC.2010.46

[5] S. Dewangan, R. S. Rao, A. Mishra and M. Gupta, "A novel approach for code smell detection: an empirical study". *IEEE Access*, Vol. 9, pp. 162 869–162 883, 2021. DOI: 10.1109/ACCESS.2021.3133810

[6] F. Pecorelli, D. Di Nucci, C. De Roover and A. De Lucia, "On the role of data balancing for machine learning-based code smell detection", In *Proceedings of the 3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation*, pp. 19–24, 2019. DOI: 10.1145/3340482.3342744.

[7] M. Hadj-Kacem and N. Bouassida, "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, 2018.

[8] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection", *Empirical Software Engineering*, Vol. 21, No. 3, pp. 1143–1191, 2016. DOI: 10.1007/s10664-015-9378-4

[9] J. Nanda and J. K. Chhabra, "SSHM: SMOTE-stacked hybrid model for improving severity classification of code smell", *International Journal of Information Technology*, Vol. 14, pp. 1–7, 2022. DOI: 10.1007/s41870-022-00943-8

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

[11] T. Sharma, V. Efstathiou, P. Louridas and D. Spinellis, "On the feasibility of transfer-learning code smells using deep learning", *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 1, pp. 1–34, 2019. DOI: 10.48550/arXiv.1904.03031

[12] N. A. A. Khleel and K. Nehéz, "Improving accuracy of code smells detection using machine learning with data balancing techniques," *The Journal of Supercomputing*, vol. 80, pp. 21 048–21 093. 2024. DOI: 10.1007/s11227-024-06265-9

[13] N. A. A. Khleel and K. Nehéz, "A novel approach for software defect prediction using CNN and GRU based on SMOTE Tomek method," *Journal of Intelligent Information Systems*, vol. 60, no. 3, pp. 673–707, Jun. 2023, DOI: 10.1007/s10844-023-00793-1

[14] N. A. A. Khleel, K. Nehéz, "Software defect prediction using a bidirectional LSTM network combined with oversampling techniques", *Cluster Comput* (2023). DOI: 10.1007/s10586-023-04170-z

[15] M. Y. Mhawish and M. Gupta, "Predicting code smells and analysis of predictions: using machine learning techniques and software metrics", *Journal of Computer Science and Technology*, Vol. 35, No. 6, pp. 1428–1445,2020. DOI: 10.1007/s11390-020-0323-7

[16] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu and L. Zhang, "Deep learning based code smell detection", *IEEE transactions on Software Engineering*, Vol. 47, No. 9, pp. 1811–1837, 2019. DOI: 10.1109/TSE.2019.2936376

[17] S. Jain and A. Saha, "Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection", *Science of Computer Programming*, Vol. 212, p. 102 713, 2021. DOI: 10.1016/j.scico.2021.102713

[18] F. Pecorelli, D. Di Nucci, C. De Roover and A. De Lucia, "A large empirical assessment of the role of data balancing in machine-learning-based code smell detection", *Journal of Systems and Software*, Vol. 169, pp. 110 693, 2020. DOI: 10.1016/j.jss.2020.110693

[19] J. Virmajoki, "Detecting code smells using artificial intelligence: a prototype", LUT-yliopisto, 2020. https://urn.fi/URN:NBN:fi-fe2020092976199

[20] D. Cruz, A. Santana and E. Figueiredo, "Detecting bad smells with machine learning algorithms: an empirical study", In *Proceedings of the 3rd International Conference on Technical Debt*, Seoul, Republic of Korea, pp. 31–40, 2020. DOI: 10.1145/3387906.3388618

Nasraldeen Alnor Adam Khleel received a BSc degree in Information Systems from the University of Kassala, Kassala-Sudan, in 2011. He got an MSc degree in Software Engineering at Khartoum University, Khartoum-Sudan, in 2015. He is currently pursuing a PhD at the University of Miskolc under the Faculty of Mechanical Engineering and Informatics, Miskolc-Hungary, since 2019. His primary research interests include Artificial Intelligence and Software Engineering.

Károly Nehéz received an MSc degree in mechanical engineering from the University of Miskolc, Hungary, in 1997 and a PhD degree in software engineering in 2003. He currently works as an associate professor at the Institute of Computer Science, head of the institute since 2019. His primary research interest is Software Engineering, although he has concurrent research in Machine Learning and Artificial Intelligence.