

Deep convolutional neural network model for bad code smells detection based on oversampling method

Nasraldeen Alnor Adam Khleel, Károly Nehéz

Institute of Information Science, University of Miskolc, Miskolc-Egyetemváros, Hungary

Article Info

Article history:

Received Mar 2, 2022

Revised Mar 25, 2022

Accepted Apr 9, 2022

Keywords:

Artificial neural networks

Code smells

Deep convolutional neural network

Software metrics

Synthetic minority over-sampling technique

ABSTRACT

Code smells refers to any symptoms or anomalies in the source code that shows violation of design principles or implementation. Early detection of bad code smells improves software quality. Nowadays several artificial neural network (ANN) models have been used for different topics in software engineering: software defect prediction, software vulnerability detection, and code clone detection. It is not necessary to know the source of the data when using ANN models but require large training sets. Data imbalance is the main challenge of artificial intelligence techniques in detecting the code smells. To overcome these challenges, the objective of this study is to presents deep convolutional neural network (D-CNN) model with synthetic minority over-sampling technique (SMOTE) to detect bad code smells based on a set of Java projects. We considered four code-smell datasets which are God class, data class, feature envy and long method and the results were compared based on different performance measures. Experimental results show that the proposed model with oversampling techniques can provide better performance for code smells detection and prediction results can be further improved when the model is trained with more datasets. Moreover, more epochs and hidden layers help increase the accuracy of the model.

This is an open access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.



Corresponding Author:

Nasraldeen Alnor Adam Khleel

Institute of Information Science, University of Miskolc

3515 Miskolc, Miskolc-Egyetemváros, Hungary

Email: nasr.alnor@uni-miskolc.hu

1. INTRODUCTION

Software systems need to be constant changed by developers. Code smells are a design issue or developer changes to source codes that indicate violation of software design rules, e.g.: abstraction or hierarchy encapsulation which can cause serious problems during systems maintenance and may impact the quality in the future. Code smells may lead to future degradation in software projects that makes software hard to evolve and maintain, and it can be an effective indication of whether source code should be refactored [1]-[3]. Detection of bad code smells in source code is a significant step for guiding the code refactoring process. Most code smells detection methods rely on object-oriented metrics as input to determine whether software projects contain bad smells. There are several case studies that have been done on object-oriented software projects to determine empirical thresholds for metrics. Several static analysis tools and code restructuring methods have been developed to discover and solve source code problems, and these tools and methods provide various ways of analyzing source codes [4]. Previous studies have classified code smells into three main categories: application, class, and method level smells [5].

Class imbalance is one of the most common problems for classification models during training and validation. The class imbalance problem hinders the efficiency of model classification and produces unbalanced

false-positive and false-negative results. There are many data sampling techniques that are used to deal with imbalanced class distributions such as oversampling and undersampling techniques. The oversampling techniques supplements instances of the minority class to the dataset, while the undersampling techniques eliminates samples of the majority class for the goal of obtaining a balanced dataset. Synthetic minority oversampling technique (SMOTE) is the most oversampling techniques that are widely used for solving class imbalance. SMOTE is technique used to increases the number of instances from the minority class by generating new synthetic instances based on the nearest neighbours belonging to that class [6]. Convolutional neural network (CNN) is the latest supervised learning approaches currently used in several practical applications such as natural language processing, image recognition, and detecting code smells [7], [8].

The main goal of this study is to introduce deep convolutional neural network model with oversampling techniques for bad code smells detection. Four different code smells were used to evaluate the capability of the proposed model based on various performance measures such as accuracy, precision, recall, and F-Measure. The structure of this paper is organized as shown in; section 2 presents a discussion on related work. Section 3 presents background on the topics of detection strategies of code smells, artificial neural networks as well as deep convolutional neural networks. After that, our research methodology is presented in section 4. Section 5 presents the experimental results and discussion followed by conclusions in the last section.

2. RELATED WORK

Research in the field of detection of bad code smells started after 1999 when Fowler *et al.* [9] specified the code smells and provided their respective refactoring opportunities in his book. There are currently several literature reviews and surveys in the field of code smell detection and refactoring [10]. We have found in the literature that there are many studies that have provided different approaches and strategies for detecting bad code smells in modern software systems [5], [11]-[13]. Kim [5] proposed a system based on a neural network model for detecting bad code smells and clarifies the relevance between code smells and object-oriented metrics. The model was evaluated based on a set of Java projects. The empirical results showed that the prediction outcomes are improved more when the model is highly trained with more datasets. Further, more epochs and hidden layers help increase the accuracy of the model. Pecorelli *et al.* [6] investigated five data balancing techniques able to mitigate data unbalancing issues to understand their impact on machine learning algorithms for code smell detection. The experiment was performed based on five code smell datasets that extracted from 13 open-source systems. The experimental results show that the machine learning models relying on SMOTE technique realize the best performance.

Virmajoki [7] presented a prototype based on machine learning, neural networks, and deep learning to detect code smells. The prototype has been implemented by using the Python programming language. The prototype was evaluated using data collected from the MLCQ code smells dataset. Although only a relatively little amount of data was collected and used for training the model, the model was able to detect code smells. Liu *et al.* [8] 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 show that the proposed approach significantly improves the state-of-the-art. Francesca Fontana *et al.* [11] presented a method using different machine learning algorithms to detect four code smells based on 74 software systems. The experimental results found that all algorithms achieved high performances, but imbalanced data caused varying performances that need to be addressed in the future studies. Kaur and Singh [13] suggested a neural network model based on object-oriented metrics for detection bad code smells. The model has been applied to find twelve bad code smells. The model has been trained and tested using many epochs and hidden layers. Experimental results showed that there is a relationship between code smells and object-oriented metrics. Sharma *et al.* [14] proposed a new method for code smell detection using CNN and recurrent neural network. The experiments were conducted based on C# sample codes. The experiment results show that, it is feasible to detect smells using DL methods and transfer-learning is possible to detect code smells with a performance similar to that of direct learning. After reviewing some previous studies, we noted that automatic detection tools are needed to help the developers to finding code smells systematic. We also noted that most of the proposed methods ignore the class imbalance problem. Therefore, our study focuses on solving the class imbalance problem using SMOTE technique.

3. BACKGROUND

In the previous studies, there are several strategies and methods developed for detecting bad code smells. This section presents a brief background information about the topics of detection strategies of code smells, artificial neural networks and as well as deep convolutional neural networks.

3.1. Detection strategies of code smells

Detection strategies are methodologies using to detecting code smells based on a combination of object-oriented metrics with predefined threshold values to identify the main symptoms that describe the code smells [15], [16]. Most approaches used to detect code smells rely on heuristics and discriminate code artifacts affected or not affected by a certain type of smell using detection rules which compare the values of pertinent metrics that extracted from source code with some experimentally identified thresholds. Where most of the current detectors need the designation of thresholds which allow them to distinguish code smells: hence, the chosen of thresholds strongly influence the performance of detectors and the chosen of appropriate representative thresholds is a key factor to compose efficient detection strategies. Previous work identified various strategies, each detection strategy is composed of a sequence of clauses connected by the logical operators AND and OR including thresholds [17].

In this study, the selected examples “smell” at the level of class, method with high frequency, that may have the greatest negative impact on the software quality, and which can be recognized by some available detection tools, at the class level selected God class and data class, while at the method level selected long method and feature envy [12]. Table 1 shows the detectors considered for building code smells datasets. Thus, the following four typical code smells were considered and evaluated in this study.

Table 1. Detectors considered for building code smells datasets

Smells	Detectors
God Class	iPlasma, PMD
Data Class	iPlasma, Fluid Tool, Antipattern Scanner
Feature Envy	iPlasma, Fluid Tool
Long Method	iPlasma, PMD, Marinescu [2]

3.1.1. God class

God classes refer to large, complex, and non-cohesive modules or classes that violate the principle of implementing only one concept per class and dominates a great part of the main system behaviour by implementing almost all the system functionalities. It is distinguished by its complexity and by encompassing a high number of instance variables and methods [18]. The detection of god class is done using three software metrics: access to foreign data (ATFD) expressing the number of foreign attributes used by a software class, weighted method per class (WMC) is the sum of all statistical complexity of all methods in software class, and tight class cohesion (TCC) refer to a relative number of method pairs of a class that accesses in common at least one feature of the measured class [19].

3.1.2. Data class

Data class is a class that has only data without functions or any behaviours, and does not process this data [6], [10], [18]. Or it is a class that passively store data [15]. This class constitutes that code smells that contain something unnecessary whose removal can make code easier to understand, effective, and cleaner. The rule to calculate data class is $(WMC < X \text{ AND } NOPA > Y)$ OR $(WMC < X \text{ AND } NOAM > Z)$, where X, Y, Z are the threshold values. WMC is the weight method counts, NOPA is the number of Public Attributes, and NOAM is the number of accessor methods [5].

3.1.3. Feature envy

Feature envy is a sign of breach of the rule of grouping behaviour with related data and happens when a method is more interested in other properties of the classes than in the ones from its class. This kind of smell affects the coupling, cohesion, and encapsulation design aspects of the system, representing a problem in the abstract design of the system. It is classified as a coupler smell and affects method/property entities [2]. Thus, this method tends to make so many calls to use the data of the other classes [5], [18].

3.1.4. Long method

The long method code smells refer to the method that is too long and increases the compatibility of the system. It is classified as a blotter smell that affects method level entities [2]. It is methods that tend to centralize the functionality of a class and tends to have too much code, to be complex, to be difficult to understand, and to use large amounts of data from other classes [11], [12].

3.2. Artificial neural networks

Artificial neural networks (ANNs) are biologically inspired computer software built to imitate the way in that the human brain processes information. ANN's are machine learning models which can be used for classification purposes. An ANNs model contains multiple units (layers) for information processing

which are known as neurons. The layers are typically named as input layer, hidden layer, and output layer [7], [20]. ANNs collect the knowledge through detecting the patterns and relationships in data and learning or training through experience. When neural networks are used for data analysis, it must be important to distinguish between ANN models which refer to the network's arrangement and ANN Algorithms which refer to computations that eventually produce the network outputs. There are two approaches to train ANNs: supervised and unsupervised. The most often used ANNs for prediction and classification tasks is a fully connected and supervised network with a backpropagation learning rule. During learning stage, weights of each neuron are considered and adjusted according to the requirements. To obtain the final weight for neurons, each neuron gives input to each preceding layer, and later these inputs are multiplied by its weight. According to this process, the neuron computes the activation level from this sum, and output is sent to the following layer where the final solution is estimated [5], [18]. The output of a neuron that is in the layer can be described by (1):

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

where Y_i represents network output, n is the total number of inputs to this neuron, X_j represents network input, W_{ij} is the connection weights between input and output nodes, b_i is the bias and f_i is the transfer function. The architecture of the neural network is shown in Figure 1.

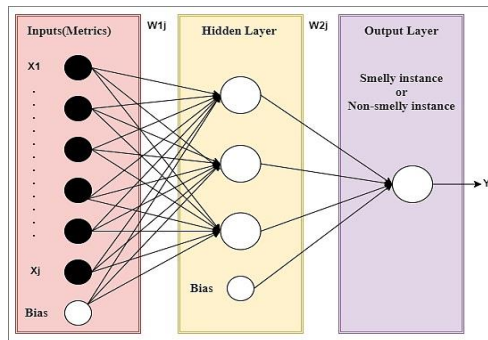


Figure 1. ANN architecture for code smells detection

3.3. Deep convolutional neural networks

Deep learning (DL) is a type of machine learning that allows computational models consisting of multiple processing layers to learn data representations with multiple levels of abstraction [14]. DL architecture has been widely used to solve many detections, classification, and prediction problems. CNN belongs to a class of deep neural networks that are used to process data that has a known, grid-like topology [21]. DL model is inspired by the typical CNN architecture used in image classification and consists of a feature extraction part and a classification part as shown in the Figure 2. These parts consist of multiple layers are convolution, batch normalization, and maximum merge layers. These layers constitute the hidden layer of the architecture. The convolutional layer performs convolution operations based on the specified filter and kernel parameters and calculates the network weights to the next layer, while the maximum pooling layer achieves a reduction in the dimension of the feature space. Batch normalization is used to mitigate the effect of different input distributions for each training mini-batch for the purpose of improving training [22], [23]. Activation functions enabling the training of DL model in a fast and accurate manner. There are many activation functions used in DL such as sigmoid, rectified linear unit (Relu) and hyperbolic tangent (Tanh) [24], [25]. Our model uses various functions such as the ReLU function as the activation function for the input and hidden layers, and the sigmoid function as the activation function for the output layer as shown in (2) and (3).

$$h_i^m = ReLU(W_i^{m-1} \times V_i^{m-1} + b^{m-1}) \quad (2)$$

where h_i^m represents convolutional layer, W_i^{m-1} represents the weights of neuron, V_i^{m-1} represents the nodes, and b^{m-1} represents the bias layer.

$$S(x) = \frac{1}{1 + e^{-\sum_k W_k x_k + b}} \quad (3)$$

where X_i represents the input, W_i is the weight of the input and b is the bias.

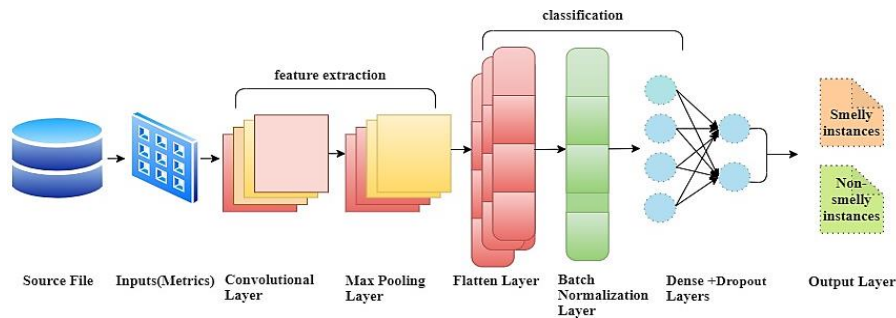


Figure 2. CNN model for code smells detection

4. METHOD

In previous work, many machine learning and ANNs algorithms have been developed for code smells detection. Most studies of code smells detection divide the data into two sets: a training set and a test set. The training set is used to train the model, whereas the testing set is used to evaluate the performance of the model. Once the model is built, its performance needs to be evaluated. This study proposed a method to train code smells detection model based on deep convolutional neural network model with oversampling techniques. The proposed method is shown in Figure 3 and Table 2 shows the components of proposed detection system.

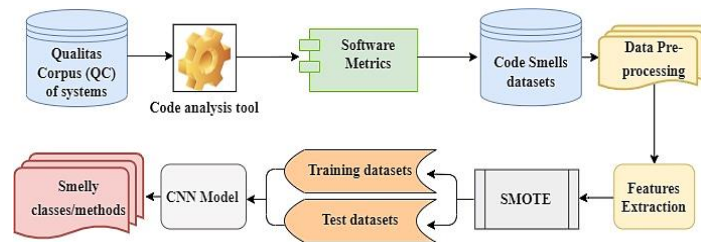


Figure 3. Proposed process of code smells detection

Table 2. Components of proposed detection system

Component name	Description
Qualitas Corpus (QC) of systems	This project composed of 111 systems written in Java
Code analysis tool	Tool used for analyze the projects to extract software metrics values
Software metrics	Metrics used for measure and characterize software engineering projects
Code Smells datasets	Set of code smells
Data Preprocessing	Improve data quality
Features extraction	Identify the most relevant features
SMOTE	Synthetic Minority Over-sampling Technique
Training datasets	Datasets used for train the model
Test datasets	Datasets used for test the model
CNN model	Model used for detecting bad code smells

5. DATA MODELLING AND COLLECTION

The code smells detection model in this study uses a supervised learning task that relies on a large set of software metrics as independent variables. Having a large number of systems or datasets is fundamental to train neural network models and allow generalization of the obtained results. To perform the analysis and experiment, the model used the proposed datasets in Fontana *et al.* [11]. The authors selected 74 open-source systems from qualitas corpus. The qualitas corpus (QC) of systems collected by Tempero *et al.* [26]. The corpus used is composed of 111 systems written in Java belonging to different application domains and characterized by different sizes. The reason for the selection is that the systems must be compliable to correctly compute the metrics values [11]. Table 3 shows a summary of the selected projects.

Table 3. Summary of project characteristics [11], [27]

Number of systems	Lines of code	Number of packages	Number of classes	Number of methods
74	6,785,568	3420	51,826	404,316

6. SOFTWARE METRICS EXTRACTION

Software metrics are essential aids to measure and improve of the software quality, and these metrics are used to measure and characterize software engineering products. The main role of software metrics is to estimate and measure some characteristics of systems such as classes, inheritance and encapsulation. Some software metrics have been used to measure software design complexity and its impact on software quality attributes such as maintainability and reusability. Other metrics have been used to solve different problems such as identifying software faults, code clone prediction, predicting testing complexity, and detect codes smells [5]. Several object-oriented metrics have been presented by Abreu, Chidamber, and Kemerer (CK), Li and Henry, MOOD, Lorenz and Kidd. These can be classified into different classes like metrics for source code analysis, metrics for software testing, and metrics for quality assurances.

There are static and dynamic metrics as well. Static metrics refer to metrics collected from the static source code like documents of specification, design schema, and code listings. For example, lines of code, weighted methods per class, and the coupling between objects. whereas dynamic metrics refer to data collected from the runtime behavior of software, e.g., dynamic coupling, dynamic lack of cohesion, and dynamic coupling between objects, these metrics aim to evaluate the design of the object-oriented application, rather than the implementation of the system [28] selected metrics in this study are a large set of object-oriented metrics that are considered as independent variables as shown in Table 4. All these metrics have been calculated through software tools, which analyzes the source code of Java projects using the eclipse JDT library. These tools are design features and metrics for Java, which have been designed to be integrated as a library into other projects [11].

Table 4. Selected metrics in this study [11]

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

7. DATA PRE-PROCESSING AND FEATURES SELECTION

Pre-processing the collected data is one of the important stages before constructing the model. Not all data collected is suitable for training and model building. Anyhow the inputs will greatly impact the performance of the model and later moreover 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 for the purpose of removing noise and unwanted outliers from the data set, dealing with missing values, and feature type conversion [14]. Feature selection (FS) is one of the significant pre-processing steps and plays a key role in classification tasks. FS is the process of identifying and removing the irrelevant and redundant features to improve the performance of the classifier. FS approaches can be divided into three main classes: wrapper-based methods, filter-based methods, and embedded methods [1], [29], [30].

8. DATA IMBALANCE

Data imbalance is one of the biggest challenges in classification models and represents cases where examples of one class are much smaller than other classes. The data imbalance problem makes classification models not effectively predict minority modules. There are many techniques that have been used to solve the problem of unbalanced data such as sampling techniques, bagging and boosting-based ensemble methods, and cost-sensitive learning techniques [6]. In this study, the dataset chosen for the task of code smells

detection is highly imbalanced. Each of the four datasets is composed of 420 instances (classes or methods), the two first datasets concern the code smells at class level where the number of instances is 140 for data class and god class. While at method level, the number of instances for Long Method is 140 and the number of instances for Feature envy is 140. To solve the problem of data imbalance, we modified the original datasets, by modifying the distribution with the algorithm of SMOTE. Figure 4 shows the distribution of learning instances over original and balanced datasets.

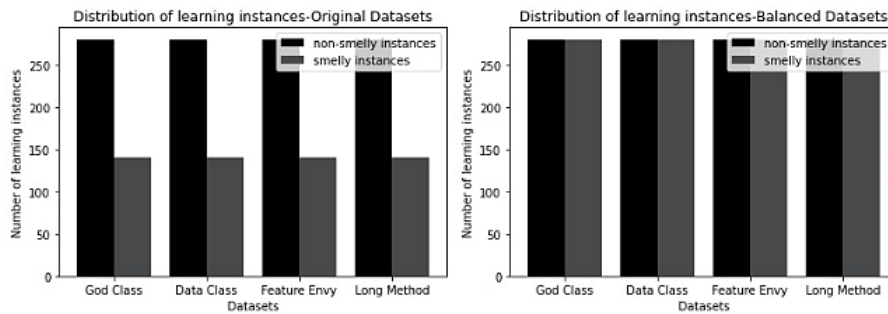


Figure 4. Distribution of learning instances over original and balanced datasets

9. MODEL BUILDING AND EVALUATION

To evaluate the model performance, the study used a set of common performance measures based on the confusion matrix such as accuracy, precision, recall, and f-measure. The confusion matrix is used to describe the performance of a classification method using a set of test data. Each row of the matrix corresponds to a predicted class, whereas each column of the matrix corresponds to an actual class. The confusion matrix is a specific table that is used to measure the performance the model. The correlation summarizes the results of the testing algorithm and present a report of i) true positive (TP), ii) false positives (FP), iii) true negatives (TN), and vi) false negatives (FN). Table 5 shows the confusion matrix.

- Accuracy=(TP+TN)/(TP+FP+FN+TN) (4)
- Precision=TP/(TP+FP) (5)
- Recall=TP/(TP + FN) (6)
- F-Measure=(2 * Recall *Precision)/(Recall + Precision) (7)

Table 5. Correlation matrix

Predicted	Actual	
	No	Yes
No	TN	FP
Yes	FN	TP

10. EXPERIMENTAL RESULTS AND DISCUSSION

An empirical study was conducted to evaluate and prove the effectiveness of our proposed model for detecting four bad code smells. In order to get accurate results, the proposed model was trained and tested with a set of huge open-source projects which contain more than 6,785,568 source code lines. The dataset size is large enough in this study to train the model. The model was trained and tested based many epochs and hidden layers. As shown in Figure 5 the LOC is the most influential metric in code smells. According to Tables 6 and 7 mentioned:

- Accuracy for the four code smell datasets: the proposed model using the balanced datasets achieves greater accuracy than the proposed model using the original datasets on the feature envy and long method datasets, which are 98% and 100%. The lowest accuracy was achieved by the proposed model using the original datasets on the Feature Envy dataset by up to 95%.
- Precision for the four code smell datasets: the proposed model using the balanced datasets achieves greater precision than the proposed model using the original datasets on the Feature Envy and Long Method datasets, which are 98% and 100%. The lowest precision was achieved by the proposed model using the original datasets on the Feature Envy and Long Method datasets by up to 93%.
- Recall for the four code smell datasets: the proposed model using the balanced datasets achieves greater recall than the proposed model using the original datasets on the god class, data class and feature envy

datasets, which are 97%, 100 % and 98%. The lowest recall was achieved by the proposed model using the original datasets on the Feature Envy dataset by up to 93%.

- F-Measure for the four code smell datasets: the proposed model using the balanced datasets achieves greater F-Measure than the proposed model using the original datasets on the god class, feature envy and long method datasets, which are 97%, 98% and 100%. The lowest F-Measure was achieved by the proposed model using the original datasets on the feature envy dataset by up to 93%.

According to Figure 6, boxplots represent performance measures obtained by the CNN model on the original and the balanced datasets. Figures 7 and 8 show the training and validation accuracy, and training and validation loss based on the results obtained from balanced datasets. After comparing the results obtained by the proposed model using the original and balanced datasets, we noticed that the best and reliable results were obtained through the proposed model using the balanced datasets by SMOTE and over-sampling techniques play an important role in dealing with problem of data imbalance.

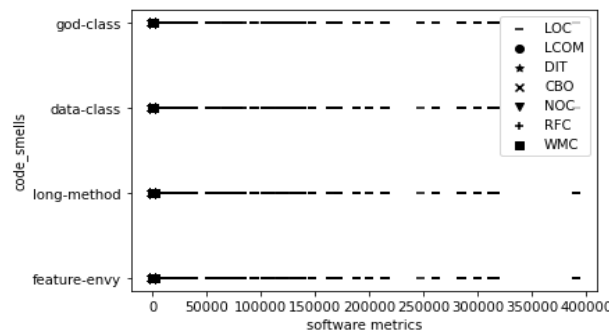


Figure 5. The distribution of software metrics with code smells

Table 6. Performance analysis for proposed CNN Model - Original Datasets

Original datasets	Performance measures			
	Accuracy	Precision	Recall	F-Measure
God class	0.96	0.97	0.94	0.96
Data class	0.99	1.00	0.96	0.98
Feature envy	0.95	0.93	0.93	0.93
Long method	0.98	0.93	1.00	0.96

Table 7. Performance analysis for proposed CNN Model - balanced datasets

Balanced datasets using SMOTE technique	Performance measures			
	Accuracy	Precision	Recall	F-Measure
God class	0.96	0.97	0.97	0.97
Data class	0.98	0.97	1.00	0.98
Feature envy	0.98	0.98	0.98	0.98
Long method	1.00	1.00	1.00	1.00

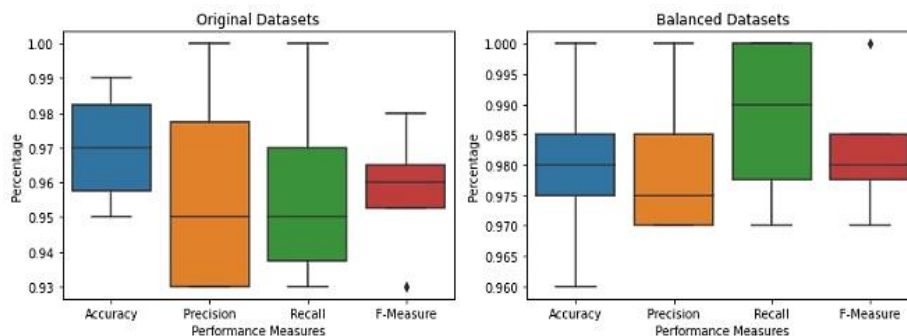


Figure 6. Boxplots represent performance measures obtained by CNN Model on the original and balanced data sets

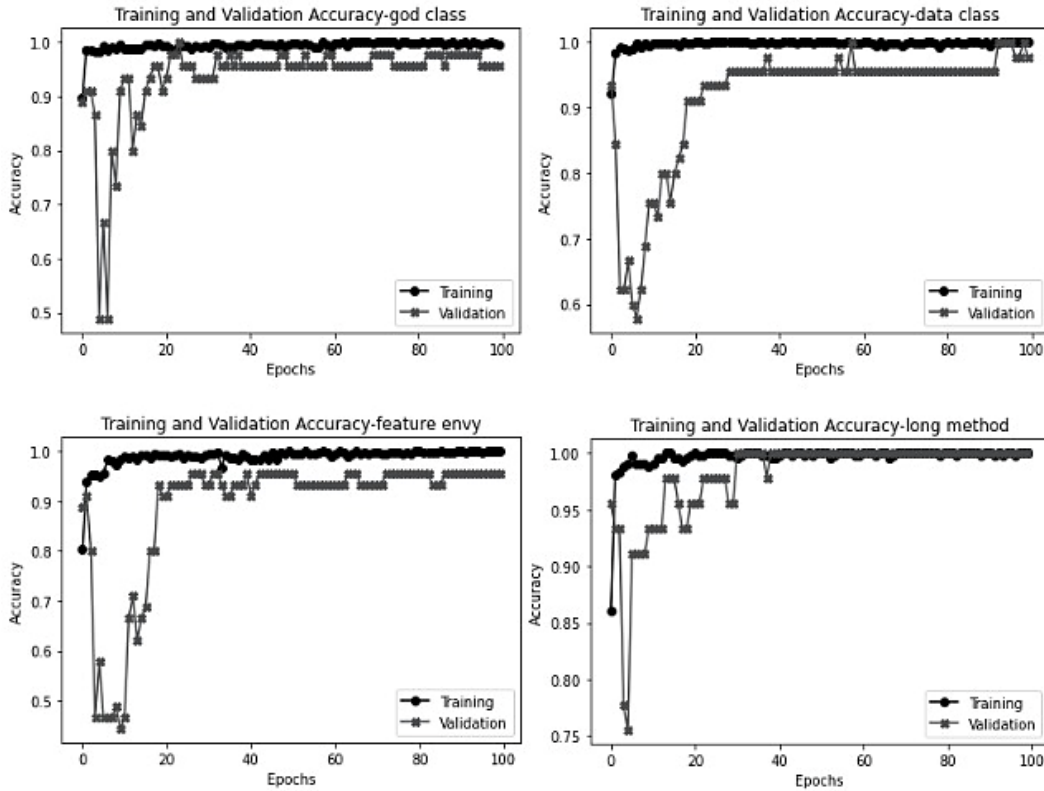


Figure 7. Training and Validation Accuracy over balanced datasets

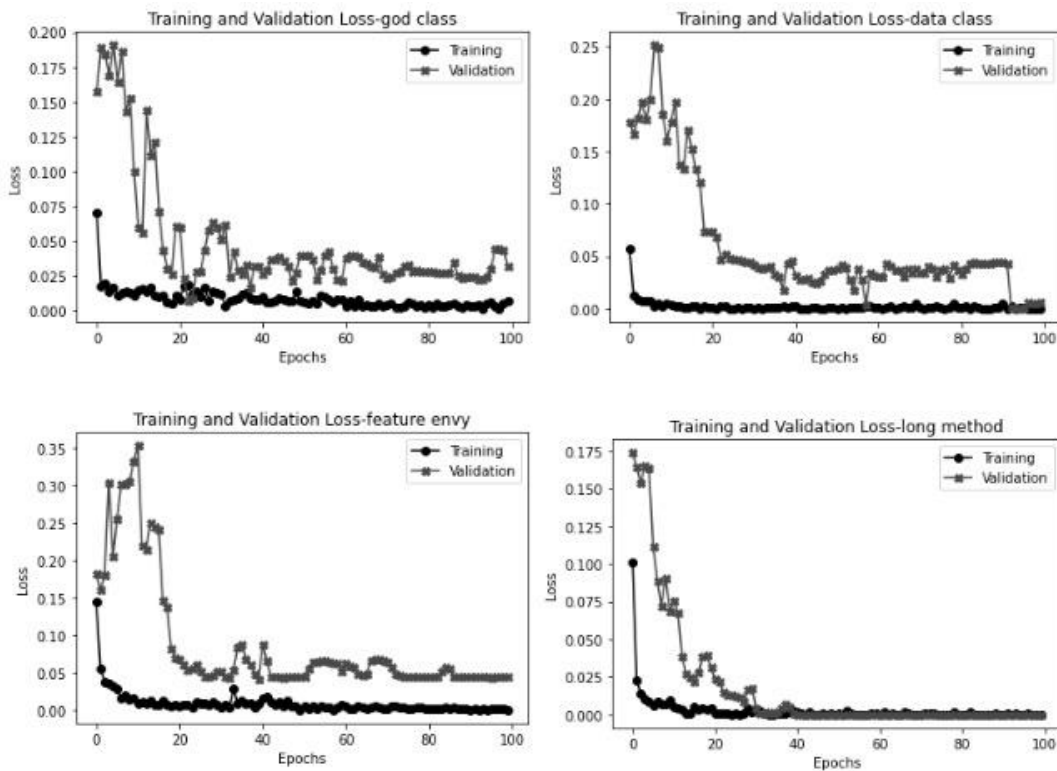


Figure 8. Training and validation loss over balanced datasets

11. CONCLUSION

This paper presents a methodology for the detection of bad code smells using CNN model with oversampling techniques based on software metrics. CNN model has been applied to sample Java projects in different application domains. In this paper, the proposed detection system attempts to detect four code smells in Java projects. To predict the performance of the model and check if the number of epochs used in training has any effect on the results, the proposed model was trained using a different number of epochs and many hidden layers. The experimental results were compared based on different performance measures such as accuracy, precision, recall, F-Measure. The results refer that, CNN model with oversampling techniques has high potential and accuracy in detecting code smells. Finally, software metrics have a close relationship with code smells.

ACKNOWLEDGEMENTS

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

REFERENCES

- [1] 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, Nov. 2020, doi: 10.1007/s11390-020-0323-7.
- [2] F. C. Luiz, B. R. De Oliveira, and F. S. Parreiras, "Machine learning techniques for code smells detection: An empirical experiment on a highly imbalanced setup," in *ACM International Conference Proceeding Series*, May 2019, pp. 1–8, doi: 10.1145/3330204.3330275.
- [3] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, Apr. 2019, doi: 10.1016/j.infsof.2018.12.009.
- [4] N. A. A. Khleel and K. Nehez, "Comprehensive study on machine learning techniques for software bug prediction," *International Journal of Advanced Computer Science and Applications*, vol. 12, no. 8, pp. 726–735, 2021, doi: 10.14569/IJACSA.2021.0120884.
- [5] D. K. Kim, "Finding bad code smells with neural network models," *International Journal of Electrical and Computer Engineering*, vol. 7, no. 6, pp. 3613–3621, Dec. 2017, doi: 10.11591/ijece.v7i6.pp3613-3621.
- [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 *MaTeSQuE 2019 - Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, co-located with ESEC/FSE 2019*, 2019, pp. 19–24, doi: 10.1145/3340482.3342744.
- [7] J. Virmajoki, "Detecting code smells using artificial intelligence: a prototype," Thesis, School of Engineering Science, Tietotekniikka, 2020.
- [8] 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, 2021, doi: 10.1109/TSE.2019.2936376.
- [9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the design of existing code addison-wesley professional," in *Berkeley, CA, USA*, 1999.
- [10] G. Saranya, H. Khanna Nehemiah, A. Kannan, and V. Nithya, "Model level code smell detection using EGAPSO based on similarity measures," *Alexandria Engineering Journal*, vol. 57, no. 3, pp. 1631–1642, Sep. 2018, doi: 10.1016/j.aej.2017.07.006.
- [11] 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, Jun. 2016, doi: 10.1007/s10664-015-9378-4.
- [12] F. Arcelli Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, Jul. 2017, doi: 10.1016/j.knosys.2017.04.014.
- [13] J. Kaur and S. Singh, "Neural network based refactoring area identification in Software System with object oriented metrics," *Indian Journal of Science and Technology*, vol. 9, no. 10, Mar. 2016, doi: 10.17485/ijst/2016/v9i10/85110.
- [14] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "On the Feasibility of transfer-learning code smells using deep learning," *arXiv preprint arXiv:1904.03031*, 2019, doi: 10.48550/arXiv.1904.03031.
- [15] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Software Quality Journal*, vol. 25, no. 2, pp. 529–552, Jun. 2017, doi: 10.1007/s11219-016-9309-7.
- [16] A. Kaur, S. Jain, S. Goel, and G. Dhiman, "A review on machine-learning based code smell detection techniques in object-oriented software system(s)," *Recent Advances in Electrical & Electronic Engineering (Formerly Recent Patents on Electrical & Electronic Engineering)*, vol. 14, no. 3, pp. 290–303, Apr. 2020, doi: 10.2174/2352096513999200922125839.
- [17] B. L. Sousa, P. P. Souza, E. M. Fernandes, K. A. M. Ferreira, and M. A. S. Bigonha, "FindSmells: Flexible composition of bad smell detection strategies," in *IEEE International Conference on Program Comprehension*, May 2017, pp. 360–363, doi: 10.1109/ICPC.2017.8.
- [18] M. Hadj-Kacem and N. Bouassida, "A hybrid approach to detect code smells using deep learning," in *ENASE 2018 - Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2018, vol. 2018-March, pp. 137–146, doi: 10.5220/0006709801370146.
- [19] M. Gradisnik, T. Beranic, S. Karakatic, and G. Mausas, "Adapting god class thresholds for software defect prediction: A case study," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2019, pp. 1537–1542, doi: 10.23919/mipro.2019.8757009.
- [20] M. S. Kadh, M. J. Mohammed, and H. Ayad, "An accurate signature verification system based on proposed HSC approach and ANN architecture," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 21, no. 1, pp. 215–223, Jan. 2021, doi: 10.11591/ijeecs.v21.i1.pp215-223.




- [21] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," *Journal of Systems and Software*, vol. 176, p. 110936, Jun. 2021, doi: 10.1016/j.jss.2021.110936.
- [22] N. Y. Abdullah, M. T. Ghazal, and N. Waisi, "Pedestrian age estimation based on deep learning," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 22, no. 3, pp. 1548–1555, Jun. 2021, doi: 10.11591/ijeecs.v22.i3.pp1548-1555.
- [23] M. A. Ramdhani, M. A. Ramdhani, D. S. adillah Maylawati, and T. Mantoro, "Indonesian news classification using convolutional neural network," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 19, no. 2, pp. 1000–1009, Aug. 2020, doi: 10.11591/ijeecs.v19.i2.pp1000-1009.
- [24] F. Barchi, E. Parisi, G. Urgese, E. Ficarra, and A. Acquaviva, "Exploration of convolutional neural network models for source code classification," *Engineering Applications of Artificial Intelligence*, vol. 97, p. 104075, Jan. 2021, doi: 10.1016/j.engappai.2020.104075.
- [25] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Sep. 2018, pp. 385–396, doi: 10.1145/3238147.3238166.
- [26] E. Tempero *et al.*, "The qualitas corpus: A curated collection of Java code for empirical studies," in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, Nov. 2010, pp. 336–345, doi: 10.1109/APSEC.2010.46.
- [27] T. Guggulothu and S. A. Moiz, "Code smell detection using multi-label classification approach," *Software Quality Journal*, vol. 28, no. 3, pp. 1063–1086, Sep. 2020, doi: 10.1007/s11219-020-09498-y.
- [28] D. I. George and P. H. Maitheen, "Analysis of object oriented metrics on a java application," *International Journal of Computer Applications*, vol. 123, no. 1, pp. 32–39, Aug. 2015, doi: 10.5120/ijca2015905226.
- [29] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?," in *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*, Mar. 2018, vol. 2018-March, pp. 602–611, doi: 10.1109/SANER.2018.8330265.
- [30] A. Alazba and H. Aljamaan, "Code smell detection using feature selection and stacking ensemble: An empirical investigation," *Information and Software Technology*, vol. 138, p. 106648, Oct. 2021, doi: 10.1016/j.infsof.2021.106648.

BIOGRAPHIES OF AUTHORS



Nasraldeen Alnor Adam Khleel    received the B.Sc. degree in Information Systems from the University of Kassala, Kassala-Sudan, in 2011. He got the M.Sc. degree in Software Engineering at Khartoum University, Khartoum-Sudan in 2015. He is currently pursuing the PhD in University of Miskolc under Faculty of Mechanical Engineering and Informatics, Miskolc-Hungary, since 2019. Research interests include Artificial Intelligence and Software Engineering. He can be contacted by email: nasr.alnor@uni-miskolc.hu.



Károly Nehéz    received the MSc degree in mechanical engineering for 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. He can be contacted by email: aitnehez@uni-miskolc.hu.