Open Journal of Applied Sciences, 2024, 14, 494-510
https://www.scirp.org/journal/ojapps
ISSN Online: 2165-3925
ISSN Print: 2165-3917

An Approach to Detect Structural Development Defects in Object-Oriented Programs

Maxime Seraphin Gnagne[1], Mouhamadou Dosso[1], Mamadou Diarra[1], Souleymane Oumtanaga[2]

[1]Department of Mathematics and Computer Science, University Félix HOUPHOUËT-BOIGNY, Abidjan, Côte d'Ivoire
[2]Research and Innovation Unit in Mathematics and Digital Sciences, National Polytechnic Institute Félix HOUPHOUËT-BOIGNY, Yamoussoukro, Côte d'Ivoire
Email: gnagnemaximeseraphin@gmail.com

How to cite this paper: Gnagne, M.S., Dosso, M., Diarra, M. and Oumtanaga, S. (2024) An Approach to Detect Structural Development Defects in Object-Oriented Programs. *Open Journal of Applied Sciences*, **14**, 494-510.
https://doi.org/10.4236/ojapps.2024.142036

Received: January 5, 2024
Accepted: February 26, 2024
Published: February 29, 2024

Abstract

Structural development defects essentially refer to code structure that violates object-oriented design principles. They make program maintenance challenging and deteriorate software quality over time. Various detection approaches, ranging from traditional heuristic algorithms to machine learning methods, are used to identify these defects. Ensemble learning methods have strengthened the detection of these defects. However, existing approaches do not simultaneously exploit the capabilities of extracting relevant features from pre-trained models and the performance of neural networks for the classification task. Therefore, our goal has been to design a model that combines a pre-trained model to extract relevant features from code excerpts through transfer learning and a bagging method with a base estimator, a dense neural network, for defect classification. To achieve this, we composed multiple samples of the same size with replacements from the imbalanced dataset MLCQ1. For all the samples, we used the CodeT5-small variant to extract features and trained a bagging method with the neural network Roberta Classification Head to classify defects based on these features. We then compared this model to RandomForest, one of the ensemble methods that yields good results. Our experiments showed that the number of base estimators to use for bagging depends on the defect to be detected. Next, we observed that it was not necessary to use a data balancing technique with our model when the imbalance rate was 23%. Finally, for blob detection, RandomForest had a median MCC value of 0.36 compared to 0.12 for our method. However, our method was predominant in Long Method detection with a median MCC value of 0.53 compared to 0.42 for RandomForest. These results suggest that the performance of ensemble methods in detecting structural development defects is dependent on specific defects.

Keywords

Object-Oriented Programming, Structural Development Defect Detection, Software Maintenance, Pre-Trained Models, Features Extraction, Bagging, Neural Network

1. Introduction

Object-oriented programming is a paradigm used by a significant number of developers in the design, development, and implementation of software systems. It facilitates the maintenance phase of developed applications. Unfortunately, this phase is increasingly jeopardized by developers introducing development defects that negatively impact software quality [1]. Poor maintenance hinders system evolution, the ease of changes that software engineers could make, program understanding, and increases the tendency for errors. In summary, poor maintenance leads to the deterioration of software quality [2] and reduces the lifespan of systems [3]. These anomalies, which are not bugs or technically incorrect codes and do not immediately disrupt program operation, indicate weaknesses in design and can slow development or increase the risk of bugs or failures in the future [4]. They are identifiable based on the taxonomy of detection approaches presented by Hadj-Kacem *et al.* [5] from four sources of information: structural, semantic, behavioral, and historical. Structural defects essentially refer to code structure that violates object-oriented design principles, such as modularity, encapsulation, data abstraction, etc. [6]. Blob and Long Method are two structural defects that align with this assertion and are widespread in source code.

They are classified according to several abstractions [4] [7] which we group into two categories: traditional heuristic approaches and machine learning-based approaches. Due to the challenges of finding threshold values for metric identification, the lack of consistency between different identification techniques, the subjectivity of developers in defining defects, and the difficulty of manually constructing optimal heuristics [8], research has shifted towards machine learning [9]. These models are mathematical techniques that use historical data to automatically identify complex patterns and make informed and intelligent decisions [10]. However, this new paradigm is mainly built by learning models taken individually, putting aside the adage that there is strength in numbers and rarely takes into account the imbalance of data in the context of source codes [10].

Knowing that pre-trained models have the ability to extract nuanced features and contextual information, they could enable better discrimination between minority and majority classes in the context of imbalanced datasets.

What machine learning method is suitable for imbalanced datasets in the context of structural development defect detection?

Our main objective is therefore to build a model based on ensemble learning whose basic estimator is composed of a pre-trained model having the capacity to contribute to the balance of classes and a classifier.

Underlying questions arise based on the above objective.

RQ1: What is the optimal number of base estimators for a model?

RQ2: Do pre-trained models alleviate class imbalance?

RQ3: Is an ensemble learning method using a pre-trained model for vector representations incorporating a deep learning classifier the state-of-the-art for ensemble methods?

To address these concerns, we organize this article as follows: In the second section, we present a literature review on detection approaches and on ensemble learning methods. Section 3 presents our approach. In Section 4, we present and discuss our results, and finally, we conclude the article.

2. Related Works

In the field of structural development defect detection approaches, several classifications have been defined [5] [11] [12]. In this work, we summarize the classification into two groups, as indicated by Yue *et al.* [13]: traditional heuristic approaches and those based on machine learning.

2.1. Traditional Heuristic Approaches

The process of heuristic approaches generally unfolds in two steps. A set of metrics associated or not with specific indicators on code instances is calculated, characterizing the considered defect. Then, thresholds are applied to these metrics [14]. Following this framework, Peldszus *et al.* [15] proposed a model that associates software metrics and various indicators of code smells to allow the system not to deteriorate as it evolves. However, the subjectivity of code smell indicators can render detection tools unusable in certain contexts.

Chen *et al.* [16] simultaneously implemented the Pysmell tool, whose detection strategy involved applying a set of metrics associated with parameterized thresholds to relevant code excerpts. Similarly, Hammad *et al.* [17] designed a plugin named JFly (Java Fly) integrated into the Eclipse environment based on a set of rules characterizing the defects to be detected, including software metrics associated with thresholds. All these approaches use threshold values, posing the recurring problem of the subjectivity of optimal choice.

Traditional heuristic approaches are increasingly abandoned in defect detection methods due, among other reasons, to the subjectivity in defining threshold values, steering research towards machine learning [9].

2.2. Machine Learning-Based Approaches

Machine learning is a field of study in artificial intelligence that aims to give machines the ability to "learn" from data through mathematical models [18]. Two methods of using machine learning algorithms represent the state of the art in research.

2.2.1. Individual Model Cases

Hamdy *et al.* [19] experimented with two recurrent neural networks, LSTM (Long short term memory) and GRU (Gated recurrent unit), and a convolutional neural network CNN to detect blob. They concluded that neural networks outperform commonly used machine learning models like Naïve Bayes, Random Forests, and Decision Trees.

Although adding the lexical and syntactic features of the source code to the software metrics has provided a set of relevant information to the training data, these features do not take into account the semantics of the code.

Kacem *et al.* [20] conducted a study using a hybrid method, coupling an unsupervised learning phase using a deep autoencoder whose purpose is to transform code snippets into vector representations of reduced dimensions and supervised learning (artificial neural network) to classify these codes based on their vector characterizations.

Sharma *et al.* [11] compared three types of models: Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and autoencoders with hidden layers composed of dense neural networks (DNN), CNN, and RNN. The authors used the open-source Tokenizer tool to generate vector representations of code.

In these two aforementioned works, the process of vector representation of code snippets does not take into account the context of the code, which defines its meaning.

To take into account the semantics of the source code, Kacem *et al.* [20] designed an approach that generates vector representations from abstract syntax trees from code excerpts used as input parameters for a variational autoencoder (VAE) whose produce semantic information through learning Finally a logistic regression classifier is applied to this semantic information to determine whether a code snippet is a defect or not. The limitation of this work lies in the procedure for extracting representative vectors from code snippets, which involves several steps, potentially making the entire system more comple.

In the search for relevant feature definitions, Škipina *et al.* [21] recommended the use of pre-trained models derived from natural language processing algorithms for source code in defining vector representations of code snippets, as opposed to metric extraction tools that take a considerable amount of time for extraction, become quickly obsolete due to the rapid evolution of language concepts, and yield inconsistent results, even for well-established code metrics. They established that the state-of-the-art in pre-trained models for vector representations in the context of code snippets is the CodeT5 model.

The use of machine learning algorithms requires transforming code snippets into vector representations. Several approaches are employed to define these representations. They range from the use of contextual and less generalizable metric extraction tools, tokenization methods that do not consider the semantic aspects of the code, vectorization from abstract syntax trees combined with machine learning, making the entire system complex, and pre-trained models. Advancements in natural language processing tasks, from which source code processing is derived through deep learning, have led researchers to utilize these methods.

2.2.2. Ensemble Learning Methods

Ensemble learning methods are meta-learning methods that combine multiple models to obtain a global and robust model. They are obtained either by using different learning algorithms, by using the same algorithm but with different parameters or initializations, or by using different training subsets with the same algorithm [22]. The most well-known ones are bagging, boosting, and stacking [23].

Khleel *et al.* [10] combined the random oversampling technique SMOTE with five machine learning models, including an XGBoost ensemble algorithm, which achieved better accuracy with both balanced and unbalanced data. Similarly, Madeyski and Lewowski [24] evaluated the performance of seven learning algorithms and an ensemble learning method to detect four structural defects. The authors concluded that, overall, the Random Forests ensemble method exhibited better performance.

The interest of these studies lies in the predominance of ensemble learning methods over individual machine learning algorithms. However, for a more thorough evaluation, a comparison between ensemble learning methods would be appropriate.

Dewangan *et al.* [25] experimented with five ensemble learning techniques and two neural networks (Dense Neural Network and Convolutional Neural Network) to assess the impact of metrics on the detection of development defects In the preprocessing phase, the authors applied the SMOTE class balancing technique to balance each class in each dataset and showed, among other things, that ensemble methods combined with the Chi-square technique, a relevant feature extraction method, improve the performance of code smell detection.

Mamatha *et al.* [26] empirically validated the effect of homogeneous ensemble methods on the prediction performance of software defect detection models. They observed a significant improvement in model performance.

To validate the best data balancing technique, Liu *et al.* [8] experimented with 31 balancing methods to detect structural defects. The authors concluded that ensemble learning techniques like DeepForest substantially improved classifier performance.

Ensemble learning techniques have proven their effectiveness in detecting structural development defects. They mitigate the class imbalance problem in certain contexts and have become the state-of-the-art methods in machine learning. However, like any machine learning algorithm, they become ineffective when the extracted features are not relevant. To the best of our knowledge, there is no approach that combines pre-trained models for relevant feature definitions and ensemble learning methods for the detection of structural defects

3. Presentation of Our Approach

In this section, we describe the targeted defects, datasets, predictors, ensemble method defining our model, algorithm, experiment parameters, and performance metrics used to evaluate our model.

3.1. Targeted Development Defects

In this article, we choose to detect the anti-pattern Blob and the code smell Long

Method. The selection of these defects is not arbitrary. Blob and Long Method are two widely prevalent structural defects in software. Code affected by these defects tends to have errors, negatively impacting software quality [2]. Given that Blob represents an anti-pattern and Long Method signifies a code smell, they can be considered representative of structural defects. Additionally, most existing datasets support these two development defects [27].

3.2. Acquisition and Processing of Training Data

In the quest for development defect detection approaches, the lack of community-validated reference data poses a challenge in validating obtained results [2]. To address this gap in reference datasets, several datasets have been proposed [1] [2] [13] [28] [29].

Lewowski and Madeyski [30] introduced MLCQ, an extensive industrial-type database annotated and validated by experienced experts.

The severity labeling of data, with a multitude of code instances, meaning a database containing both defect and non-defect instances, advocate for the MLCQ dataset. Consequently, we choose the MLCQ dataset based on the aforementioned considerations. We define two classes, expressed as [31] as follows:

$$C_0 = \left\{ x_i \,/\, severity\left(x_i\right) = \text{"none"} \right\}$$

$$C_1 = \left\{ x_i \,/\, severity\left(x_i\right) \in \left\{ \text{minor}, \text{major}, \text{critical} \right\} \right\}$$

where x_i is a code snippet at the class or method level.

In this study, we will use two reduced MLCQ datasets, one balanced and another imbalanced, to account for real proportions for a proper evaluation of our model. Table 1 and Table 2 provide the distributions of these two datasets.

3.3. Vector Representation and Class Imbalance Management

We employ the pre-trained CodeT5 model in the context of this work for generating code vector representations. This choice is based on its state-of-the-art

Table 1. Dataset containing Blob.

Development defects	Number of blobs	Blob rate	Number of Instances negatives	Instances negatives rate	Total
Balanced set	104	50%	104	50%	208
Unbalanced set	104	23%	348	77%	452

Table 2. Dataset containing long method.

Development defects	Number of Long Method	Long Method rate	Number of Instances negatives	Instances negatives rate	Total
Balanced set	140	50%	140	50%	280
Unbalanced set	140	23%	460	77%	600

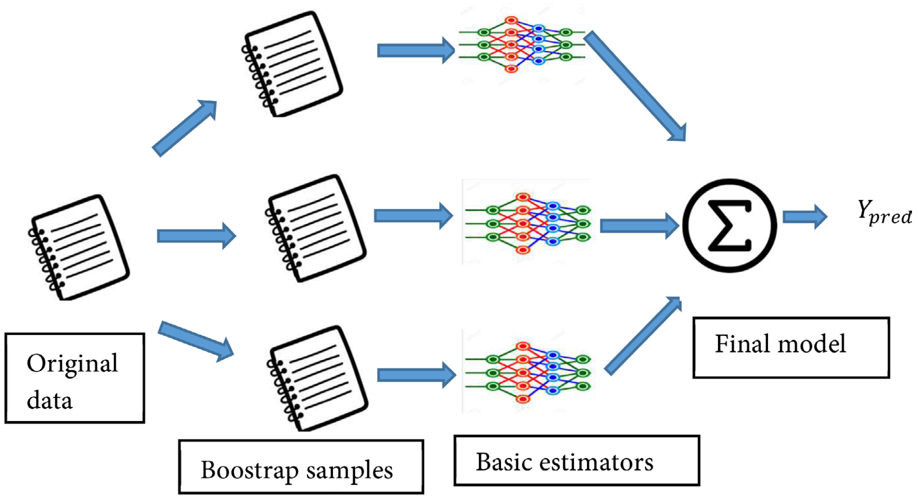


Figure 1. The bagging technique.

performance, and it captures the semantic nuances necessary for detecting code smells [21]. Leveraging the pre-training of this model, we address the imbalance issue within our dataset.

3.4. Ensemble Learning Methods

The bagging model is formalized as follows: Let $Z = \{(x_1, y_1), \cdots, (x_n, y_n)\}$ be the initial sample, where x_i is a code snippet, and $y_i \in \{0;1\}$. Let B be bootstrap samples of n observations: $Z^b = \{(x_1^b, y_1^b), \cdots, (x_n^b, y_n^b)\}$, $b = 1, \cdots, B$.

Let $\hat{f}_b(.) = DNN_b \circ MOD\_PRE_b$, where $\hat{f}_b(.)$ is the model trained with bootstrap sample *b*, composed of the pre-trained model MOD_PRE followed by a DNN.

The ensemble model is defined as:

$$\hat{f}(.) = argmax_{i=1}^{B} \{\hat{f}_b(.)\} \tag{1}$$

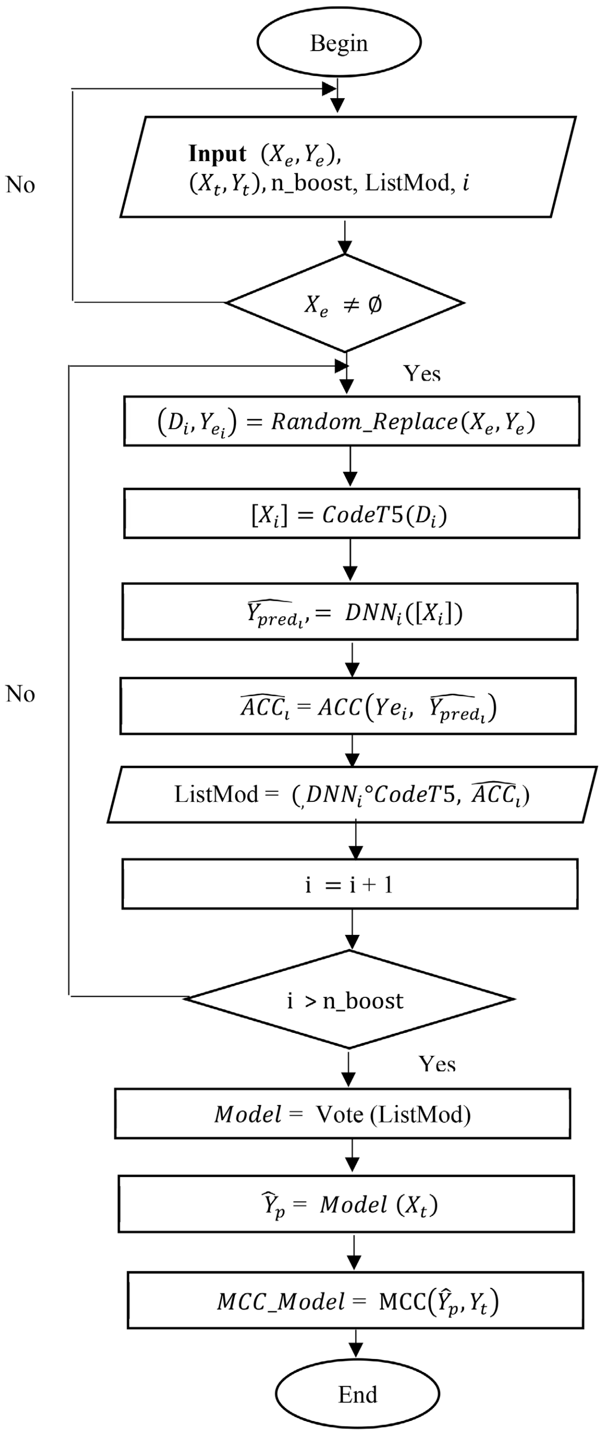
In our work, we experiment with CodeT5 as a pre-trained model and Roberta Classification Head for the DNN. The bagging technique is illustrated in the Figure 1.

3.5. Our Algorithm

3.5.1. Description of Our Algorithm

(X_e, Y_e), and (X_t, Y_t) are respectively the training and test data, ListMod is the container for the trained base estimators andn_boostcorresponds to the number of base estimators to train. For each iteration i of n_boost, we select a sample (D_i, Y_{e_i}) with replacement using the Random_replace function of the same size as the initial dataset X_e. We extract the features $[X_i]$ from the code snippets using the pre-trained CodeT5 model, the first component of our model. This vector representation $[X_i]$ is used as input to train our base estimator, which is the dense neural network DNN_i.

ACC is the function that evaluates the accuracy of the base estimator.Once all the base estimators are trained, each is associated with an accuracy. We obtain a list of pairs (DNN_i , $\widehat{ACC_i}$) that will be subjected to a Majority Voting procedure. This is the task of the vote function. It returns the final model that will be used for classification. Model evaluation is performed using the Matthews Correlation Coefficient (MCC) metric adapted for imbalanced data.We illustrate our algorithm with the diagram below.


Begin
Input (X_e, Y_e), (X_t, Y_t), n_boost, ListMod, i
$X_e \neq \emptyset$
No
Yes
$(D_i, Ye_i) = Random\_Replace(X_e, Y_e)$
$[X_i] = CodeT5(D_i)$
$\widehat{Ypred_i} = DNN_i([X_i])$
$\widehat{ACC_i} = ACC(Ye_i, \widehat{Ypred_i})$
ListMod = $(DNN_i \circ CodeT5, \widehat{ACC_i})$
i = i + 1
i > n_boost
No
Yes
$Model$ = Vote (ListMod)
$\hat{Y}_p = Model\ (X_t)$
MCC_Model = MCC$(\hat{Y}_p, Y_t)$
End

3.5.2. Explanation of the Algorithm

(X_e, Y_e) and (X_t, Y_t) are respectively the training and test data, ListMod is the container for the trained base estimators and n_boost corresponds to the number of base estimators to train. For each iteration *i* of n_boost, we select a sample (D_i, Y_{e_i}) with replacement using the Random_replace function of the same size as the initial dataset X_e. We extract the features $[X_i]$ from the code snippets using the pre-trained CodeT5 model, the first component of our model. This vector representation $[X_i]$ is used as input to train our base estimator, which is the dense neural network DNN_i.

ACC is the function that evaluates the accuracy of the base estimator.

Once all the base estimators are trained, each is associated with an accuracy. We obtain a list of pairs (DNN_i, $\widehat{ACC_i}$) that will be subjected to a Majority Voting procedure. This is the task of the vote function. It returns the final model that will be used for classification. Model evaluation is performed using the Matthews Correlation Coefficient (MCC) metric adapted for imbalanced data.

3.6. Performance Evaluation Metrics

The usual model evaluation metrics that we will present subsequently are deduced from the confusion matrix defined by Table 3 for a binary classification.

Accuracy is the proportion of correctly predicted instances compared to all instances

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FN} + \text{FP}} \tag{2}$$

Precision, also called positive predictive value, is the ratio of the number of true positives to the total number of positive predictions.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3}$$

Recall or sensitivity or true positive rate is the proportion of positive instances actually predicted by the model among all truly positive instances

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4}$$

The F1-score is the harmonic mean of precision and recall

$$\text{F1-score} = \frac{2 * P * R}{P + R} \tag{5}$$

Mattheus Correlation Coefficient (MCC) measures the quality of a classification by comparing the model's predictions with the true class labels. It is particularly

Table 3. Confusion matrix for binary classification.

	Positive predicted	Negative predicted
Real positive	True Positive (TP)	False Negative (FN)
Real negative	False Positive (FP)	True Negative (TN)

useful in scenarios where classes are unbalanced [32].

$$\text{MCC} = \frac{\text{TP} * \text{TN} - \text{FP} * \text{FN}}{\sqrt{(\text{TP}+\text{FP})(\text{TP}+\text{FN})(\text{TN}+\text{FP})(\text{TN}+\text{FN})}} \quad (6)$$

4. Results and Discussion

4.1. Results

This section presents the results of our experiments in accordance with our research questions. In section 4.1.1, we will evaluate the impact of the number of base estimators in ensemble methods. Section 4.1.2 will justify the necessity or otherwise of class balancing through additional techniques. Section 4.1.3 will compare our method to RandomForest, and finally, section 3.4.2 will discuss our results.

4.1.1. Evaluation of the Number of Base Estimators in Ensemble Models

To better understand the impact of the number of base estimators in ensemble methods, we compare our model and RandomForest by varying the number of base estimators on two types of datasets. One balanced and the other imbalanced. Table 4 and Table 5, depicted by Figure 2 and Figure 3, respectively, show a comparison of the average accuracies between our model and RandomForest on

Table 4. Comparative table of average Accuracies between our model and RandomForest in the detection of Blob on the balanced dataset.

Number of estimators / Number of trials	2	4	6	8	10
RandomForest	0.63	0.63	0.64	0.64	0.63
Our model	0.59	0.64	0.65	0.65	0.67

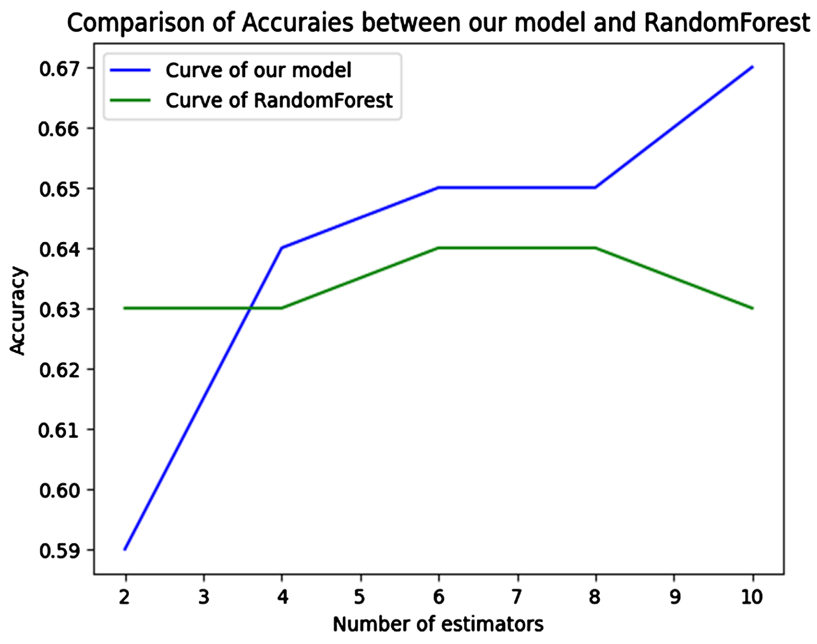


Figure 2. Comparison of accuracies in Blob detection on the balanced dataset.

Table 5. Comparative table of average Accuracies between our model and RandomForest in the detection of LongMethod on the balanced dataset.

Number of trials \ Number of estimators	2	4	6	8	10
RandomForest	0.61	0.61	0.67	0.6	0.59
Our model	0.63	0.61	0.60	0.61	0.60

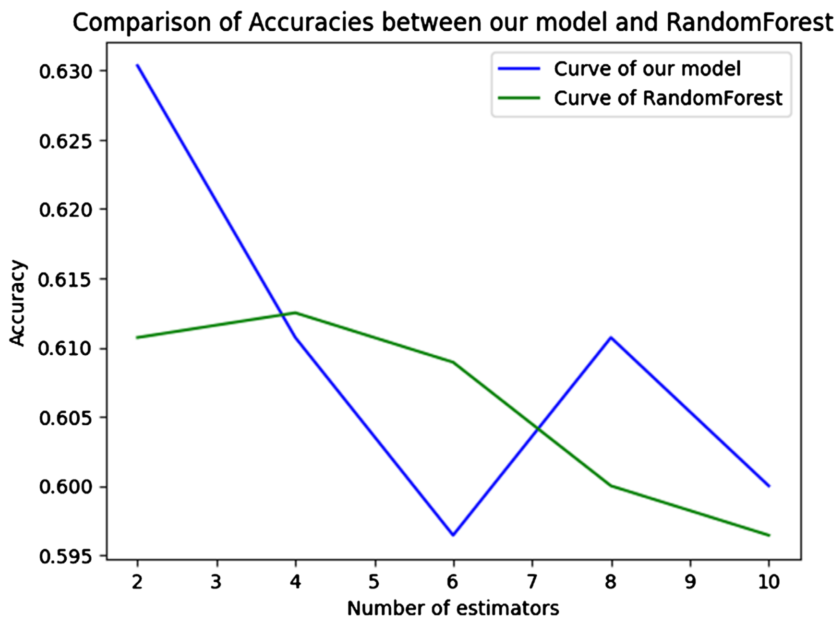


Figure 3. Comparison of accuracies in Blob detection on the balanced dataset.

Table 6. Comparative table of average MCCs between our model and RandomForest in the detection of LongMethod on the unbalanced dataset.

Number of trials \ Number of estimators	2	4	6	8	10
RandomForest	0.42	0.47	0.50	0.51	0.52
Our model	0.49	0.52	0.53	0.54	0.53

the balanced dataset. Meanwhile, Table 6 and Table 7, illustrated by Figure 4 and Figure 5, present the average MCCs between our model and RandomForest on the imbalanced dataset.

4.1.2. Class Balancing

Can the use of pre-trained models adjust to minority classes? To address this concern, we evaluated our model with and without the SMOTE technique. On average, we achieved 87% accuracy, whether we associated the SMOTE technique with our model or not, with a very slight increase of 3% in MCC in favor of the balancing technique in the first eight trials. These empirical results suggest the unnecessary use of balancing techniques when pre-trained models are components of the model.

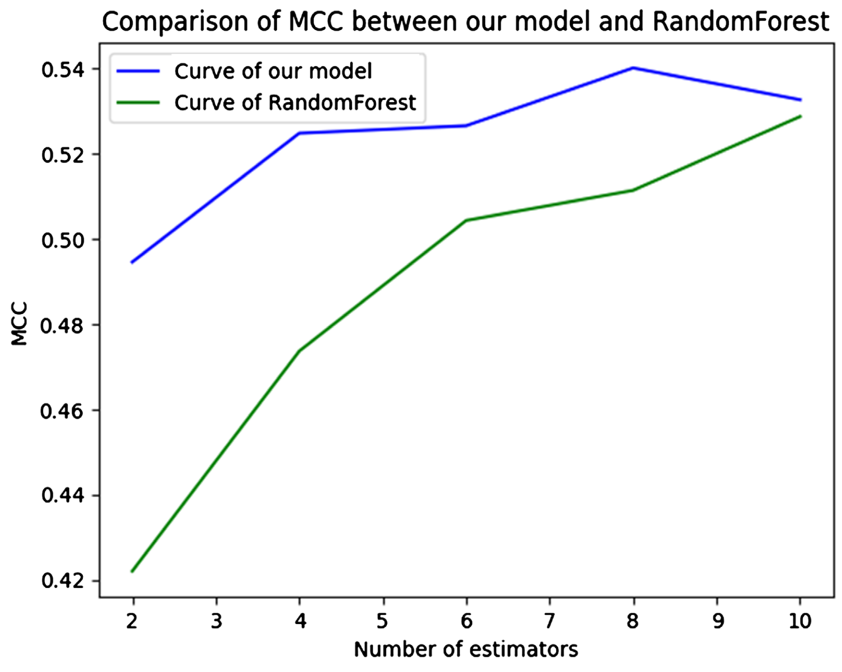


Figure 4. Comparison of MCCs in LongMethod detection on the unbalanced dataset.

Table 7. Comparative table of average MCCs between our model and RandomForest in the detection of blob on the unbalanced dataset.

Number of estimators / Number of trials	2	4	6	8	10
RandomForest	0.20	0.21	0.22	0.24	0.22
Our model	0.13	0.16	0.21	0.13	0.10

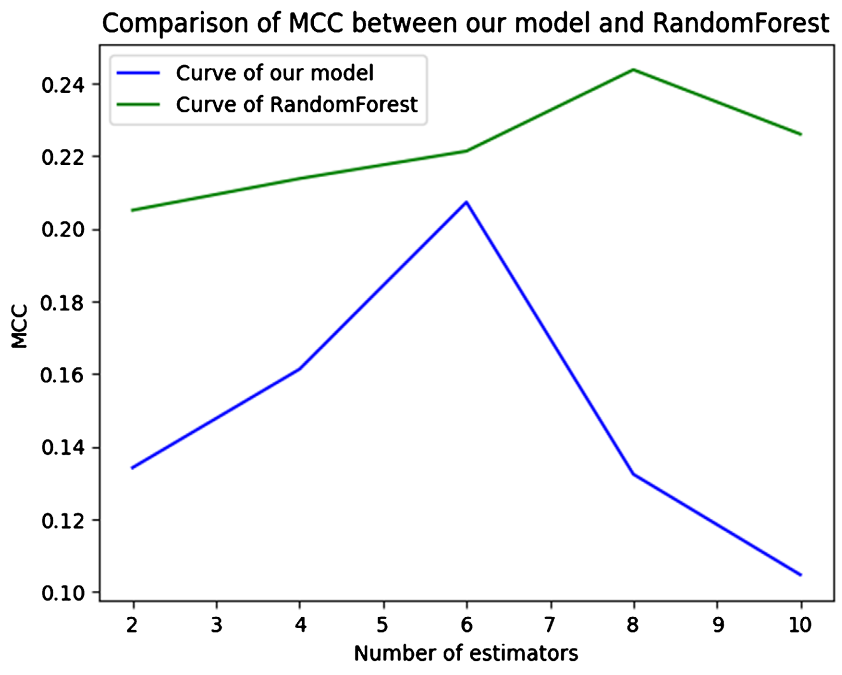


Figure 5. Comparison of MCCs in Blob detection on the unbalanced dataset.

4.1.3. Comparison of Our Model and RandomForest

Our approach is compared to that of [24], specifically to the RandomForest algorithm, which showed better performance according to the authors. Our 20 experiments rely on 10 base estimators trained on imbalanced datasets. We aggregated precision and recall by the mean, F1-score using equation (5), and accuracy and MCC by the median.

We summarize the results obtained in Table 8 and Table 9 with the associated graphical representations in Figure 6 and Figure 7, respectively.

4.2. Discussion

We posed questions for which experiments were undertaken to empirically confirm or refute assertions.

RQ1: How many basic estimators are needed for an optimal model?

To determine this number, we compared our model to RandomForest by varying the number of estimators. We observed that for a number of estimators

Table 8. Evaluation of our model and RandomForest in the detection of Long Method.

	Precision	Recall	F1-score	Accuracy	MCC
RandomForest	0.75	0.68	0.71	0.77	0.42
Our Model	0.80	0.73	0.76	0.87	0.53

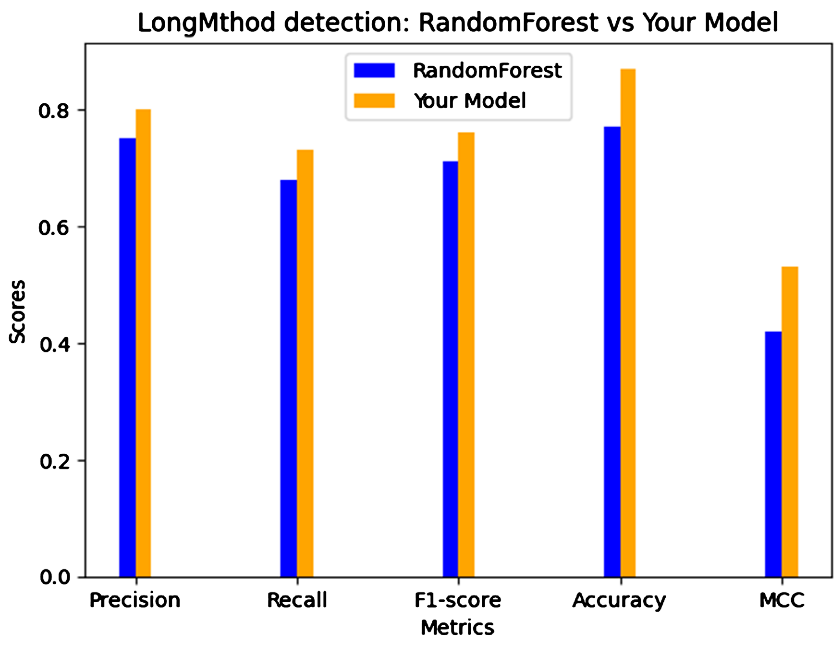


Figure 6. Evaluation of our model and RandomForest in the detection of Long Method.

Table 9. Evaluation of our model and RandomForest in the detection of Blob.

	Precision	Recall	F1-score	Accuracy	MCC
RandomForest	0.68	0.69	0.68	0.73	0.36
Our Model	0.59	0.56	0.57	0.68	0.12

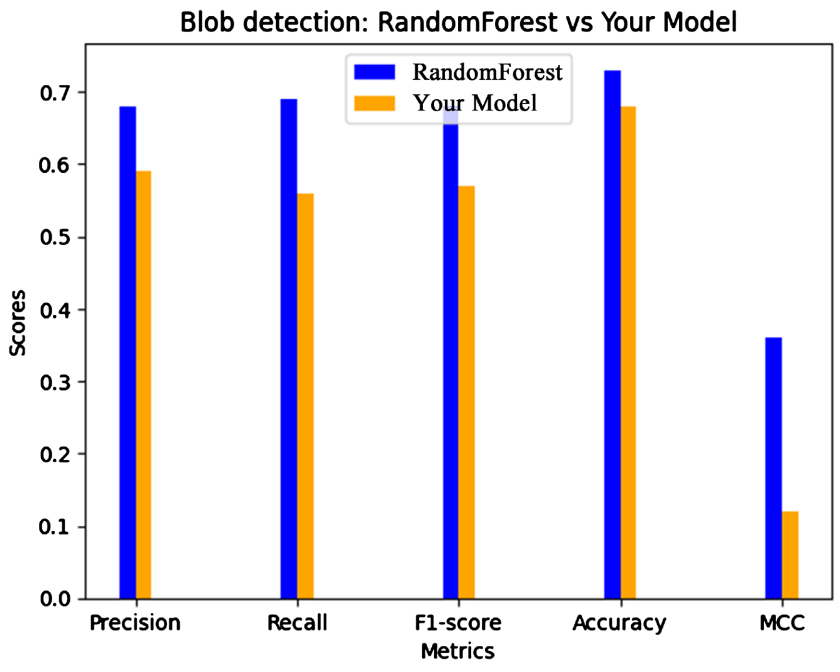


Figure 7. Evaluation of our model and RandomForest in the detection of Blob.

less than or equal to 20 for balanced datasets with a maximum size of 140, our model performed well in terms of accuracy. However, when data classes became imbalanced with a maximum size of 600, the number of estimators allowed by our experimental conditions became less than or equal to 10. In this context, we observed that RandomForest better detects the Blob anti-pattern from 7 estimators, while our model becomes effective from 10 estimators in detecting the LongMethod. These results need verification for larger numbers of estimators and larger datasets. We observed that when the imbalance is significant, both models tend to overfit. This observation could be verified by reconsidering the data and base estimator sizes.

RQ2: Do pre-trained models mitigate class imbalance?

From the experiments, it is not necessary to use additional class balancing techniques when pre-trained models are used to transform code snippets into vector representations. However, the question remains open when the imbalance is significant. We only used 8 experiments, as we noted that beyond this point, the variation in accuracy became low. This observation needs further justification.

RQ3: Is an ensemble method using a pre-trained model for vector representations and incorporating a deep learning classifier the state of the art in ensemble methods?

Since we use an unbalanced set, the preferred metric is MCC.

In Blob detection, the median MCC of our method is almost 20% lower than that of RandomForest. The F1-score and other performance metrics confirm the predominance of RandomForest over our model (See graph 4). However, our model outperforms RandomForest in LongMethod detection with a median MCC equal to 0.53 compared to 0.42 for RandomForest (See graph 3). These results suggest that ensemble learning methods based on deep learning estimators are not inherently more dominant in defect detection. However, the work of [19] has shown that neural networks outperform non-deep learning methods. Our underperformance in Blob detection could be explained by the small size of our dataset, as neural networks typically perform better when trained on large datasets.

5. Conclusions

Structural defects have a negative impact on software quality, making software maintenance challenging. Existing approaches do not integrate pre-trained models, which have the ability to define vector representations that consider the code semantics, and neural networks, which are highly effective in classification tasks. Therefore, we designed a bagging approach with CodeT5 and a dense neural network as the base estimator to detect Blob and LongMethod. We aimed to determine the optimal number of base estimators, but due to hardware resource limitations, we couldn't establish a threshold for the number of estimators. In this study, we demonstrated that pre-trained models could, to some extent, address the issue of class imbalance in data. We compared our model with RandomForest, a reference model. Our results suggested that the choice of methods for detecting structural defects depended on the type of defects. However, we can confirm, based on the work of [18], that the size of the data used biased the results.

For future work, we will use larger datasets and a greater number of estimators to evaluate our model. We will also examine the complexity of our model to reduce execution time compared to RandomForest.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Sharma, T. and Kessentini, M. (2021) Qscored: A Large Dataset of Code Smells and Quality Metrics. 2021 *IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, Madrid, 17-19 May 2021, 590-594. https://doi.org/10.1109/MSR52588.2021.00080

[2] Kovačević, A., *et al.* (2022) Automatic Detection of Long Method and God Class Code Smells through Neural Source Code Embeddings. *Expert Systems with Applications*, **204**, Article ID: 117607. https://doi.org/10.1016/j.eswa.2022.117607

[3] Mhawish, M.Y. and Gupta, M. (2020) Predicting Code Smells and Analysis of Predictions: Using Machine Learning Techniques and Software Metrics. *Journal of Computer Science and Technology*, **35**, 1428-1445. https://doi.org/10.1007/s11390-020-0323-7

[4] Velioğlu, S. and Selçuk, Y.E. (2017) An Automated Code Smell and Anti-Pattern Detection Approach. 2017 *IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*, London, 7-9 June 2017, 271-275. https://doi.org/10.1109/SERA.2017.7965737

[5] Hadj-Kacem, M. and Bouassida, N. (2018) Towards a Taxonomy of Bad Smells Detection Approaches. *Proceedings of the* 13*th International Conference on Software Technologies*, Vol. 1, 164-175. https://doi.org/10.5220/0006869201980209

[6] Gupta, A., Suri, B. and Misra, S. (2017) A Systematic Literature Review: Code Bad Smells in Java Source Code. In: Gervasi, O., Murgante, B., Misra, S., *et al*., Éds., *Computational Science and Its Applications—ICCSA* 2017, Lecture Notes in Computer Science, Vol. 10408, Springer International Publishing, Cham, 665-682. https://doi.org/10.1007/978-3-319-62404-4_49

[7] Sharma, T. and Spinellis, D. (2018) A Survey on Software Smells. *Journal of Systems and Software*, **138**, 158-173. https://doi.org/10.1016/j.jss.2017.12.034

[8] Liu, H., Jin, J., Xu, Z., Zou, Y., Bu, Y. and Zhang, L. (2019) Deep Learning Based Code Smell Detection. *IEEE Transactions on Software Engineering*, **47**, 1811-1837.

[9] Hadj-Kacem, M. and Bouassida, N. (2018) A Hybrid Approach to Detect Code Smells Using Deep Learning. *Proceedings of the* 13*th International Conference on Evaluation of Novel Approaches to Software Engineering ENASE*, **1**, 137-146. https://doi.org/10.5220/0006709801370146

[10] Khleel, N.A.A. and Nehéz, K. (2023) Detection of Code Smells Using Machine Learning Techniques Combined with Data-Balancing Methods. *International Journal of Advances in Intelligent Informatics*, **9**, 402-417. https://doi.org/10.26555/ijain.v9i3.981

[11] Sharma, T., Efstathiou, V., Louridas, P. and Spinellis, D. (2021) Code Smell Detection by Deep Direct-Learning and Transfer-Learning. *Journal of Systems and Software*, **176**, Article ID: 110936. https://doi.org/10.1016/j.jss.2021.110936

[12] Alazba, A. and Aljamaan, H. (2021) Code Smell Detection Using Feature Selection and Stacking Ensemble: An Empirical Investigation. *Information and Software Technology*, **138**, Article ID: 106648. https://doi.org/10.1016/j.infsof.2021.106648

[13] Yu, X., Li, F., Zou, K., Keung, J., Feng, S. and Xiao, Y. (2023) On the Relative Value of Imbalanced Learning for Code Smell Detection. https://doi.org/10.22541/au.167338512.23766841/v1

[14] Azeem, M.I., Palomba, F., Shi, L. and Wang, Q. (2019) Machine Learning Techniques for Code Smell Detection: A Systematic Literature Review and Meta-Analysis. *Information and Software Technology*, **108**, 115-138. https://doi.org/10.1016/j.infsof.2018.12.009

[15] Peldszus, S., Kulcsár, G., Lochau, M. and Schulze, S. (2016) Continuous Detection of Design Flaws in Evolving Object-Oriented Programs Using Incremental Multi-Pattern Matching. *Proceedings of the* 31*st IEEE*/*ACM International Conference on Automated Software Engineering*, Singapore, 3-7 September 2016, 578-589. https://doi.org/10.1145/2970276.2970338

[16] Chen, Z., Chen, L., Ma, W. and Xu, B. (2016) Detecting Code Smells in Python Programs. 2016 *IEEE International Conference on Software Analysis*, *Testing and Evolution* (*SATE*), Kunming, 3-4 November 2016, 18-23. https://doi.org/10.1109/SATE.2016.10

[17] Hammad, M. and Labadi, A. (2016) Automatic Detection of Bad Smells from Code Changes. *International Review on Computers and Software*, **11**, 1016-1027. https://doi.org/10.15866/irecos.v11i11.10590

[18] Apprentissage Automatique. https://www.cnil.fr/fr/definition/apprentissage-automatique

[19] Hamdy, A. and Tazy, M. (2020) Deep Hybrid Features for Code Smells Detection. *Journal of Theoretical and Applied Information Technology*, **98**, 2684-2696.

[20] Hadj-Kacem, M. and Bouassida, N. (2019) Deep Representation Learning for Code Smells Detection Using Variational Auto-Encoder. 2019 *International Joint Conference on Neural Networks* (*IJCNN*), Budapest, 14-19 July 2019, 1-8. https://doi.org/10.1109/IJCNN.2019.8851854

[21] Škipina, M., Slivka, J., Luburić, N. and Kovačević, A. (2022) Automatic Detection of Feature Envy and Data Class Code Smells Using Machine Learning. https://doi.org/10.36227/techrxiv.21732059.v1

[22] Dong, Y. (2013) Modélisation probabiliste de classifieurs d'ensemble pour des problèmes à deux classes. THESE pour l'obtention du grade de DOCTEUR, Université de Technologie Troyes, Troyes.

[23] Khan, A.A., Chaudhari, O. and Chandra, R. (2023) A Review of Ensemble Learning and Data Augmentation Models for Class Imbalanced Problems: Combination, Implementation and Evaluation. *Expert Systems with Applications*, **244**, Article ID: 122778. https://doi.org/10.1016/j.eswa.2023.122778

[24] Madeyski, L. and Lewowski, T. (2023) Detecting Code Smells Using Industry-Relevant Data. *Information and Software Technology*, **155**, Article ID: 107112. https://doi.org/10.1016/j.infsof.2022.107112

[25] Dewangan, S., Rao, R.S., Mishra, A. and Gupta, M. (2022) Code Smell Detection Using Ensemble Machine Learning Algorithms. *Applied Sciences*, **12**, Article No. 10321. https://doi.org/10.3390/app122010321

[26] Mamatha, R., Kumari, P.L.S. and Sharada, A. (2024) Enhanced Software Defect Prediction through Homogeneous Ensemble Models. *International Journal of Intelligent Systems and Applications in Engineering*, **12**, 676-684.

[27] Zakeri-Nasrabadi, M., Parsa, S., Esmaili, E. and Palomba, F. (2023) A Systematic Literature Review on the Code Smells Datasets and Validation Mechanisms. *ACM Computing Surveys*, **55**, Article No. 298. https://doi.org/10.1145/3596908

[28] Fontana, F.A., Ferme, V., Zanoni, M. and Roveda, R. (2015) Towards a Prioritization of Code Debt: A Code Smell Intensity Index. 2015 *IEEE 7th International Workshop on Managing Technical Debt* (*MTD*), Bremen, 2 October 2015, 16-24. https://doi.org/10.1109/MTD.2015.7332620

[29] Arcelli Fontana, F., Mäntylä, M.V., Zanoni, M. and Marino, A. (2016) Comparing and Experimenting Machine Learning Techniques for Code Smell Detection. *Empirical Software Engineering*, **21**, 1143-1191. https://doi.org/10.1007/s10664-015-9378-4

[30] Lewowski, T. and Madeyski, L. (2022) Code Smells Detection Using Artificial Intelligence Techniques: A Business-Driven Systematic Review. *Developments in Information & Knowledge Management for Business Applications*, **3**, 285-319. https://doi.org/10.1007/978-3-030-77916-0_12

[31] Madeyski, L. and Lewowski, T. (2020) MLCQ: Industry-Relevant Code Smell Data Set.

[32] Chicco, D. and Jurman, G. (2020) The Advantages of the Matthews Correlation Coefficient (MCC) over F1 Score and Accuracy in Binary Classification Evaluation. *BMC Genomics*, **21**, Article No. 6. https://doi.org/10.1186/s12864-019-6413-7