Knowledge-Based Systems 286 (2024) 111390

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

CBReT: A Cluster-Based Resampling Technique for dealing with imbalanced data in code smell prediction

Praveen Singh Thakur *, Mahipal Jadeja, Satyendra Singh Chouhan

Department of CSE, MNIT Jaipur, 302017, India

ARTICLE INFO

Dataset link: https://github.com/hjamaan/IST2021-CodeSmellStackingEnsemble/tree/main/Datasets/Original, https://codeocean.com/capsule/5256791/tree/v1l

Keywords:
Code smell prediction
Imbalance learning
Oversampling
Software maintainability
Empirical study

ABSTRACT

Code smell refers to substandard design patterns in software's source code that may lead to faults-prone implementation. Machine learning-based code smell prediction models suffer from data imbalance problems, i.e., one class contains significantly more instances than another. The existing oversampling approaches, such as SMOTE (Synthetic Minority Over-sampling Technique), have been used for balancing the code smell dataset by generating synthetic samples for the minority class. However, the distribution of classes of code smell datasets is overlapped; hence, randomly generated instances can damage the decision boundary between both classes. This paper addresses this issue and proposes a novel Cluster-Based Resampling Technique, *CBReT*, that generates synthetic instances by considering the distribution of the code smell data. The *CBReT* first formulates clusters (containing minority and majority instances) based on the data distribution using Gaussian Mixture Model (GMM). Next, each cluster is balanced separately by synthesizing minority instances. While balancing the clusters, the *CBReT* also checks the validity of the synthetic instances so that each synthetic instance holds similar properties as the other minority instances. To assess the performance of *CBReT*, extensive experiments have been conducted on the four publicly available benchmark code smell datasets. We have used various performance metrics to evaluate our model's performance. The experimental results show that the *CBReT* technique significantly increased the performance of the code smell prediction model by 0.18% (min) and 9.08% (max) compared to the state-of-the-art imbalance learning approaches.

1. Introduction

Code smell is not an error or bug; it is a symptom that occurs due to poor implementation design of the software. A project containing code smell can give correct output, but it increases the effort in maintenance. Due to that, in the future, code smell can cause software failure [1]. The code smell problem occurs when the programmer does not follow the proper programming standard. In previous studies, it has been found that code smell negatively impacts the developer's understanding of maintenance tasks and the system's design [2–4]. An example of a type of code smell (Feature Envy) is shown in Fig. 1, where we can see that the getAddress method is associated with the ContactInfo class but defined inside the User class.

The researchers have developed many automatic code smell detectors (tool-based and Machine Learning-based) to detect the types of code smells. The tool-based detectors use different heuristic algorithms to detect code smells by calculating the metrics with a threshold value to differentiate the smelly and non-smelly items [5,6]. The threshold value plays a crucial role in the performance of the detectors. The detector's performance will decrease if the threshold value is not precise. The threshold value can vary for the different types of code smells. Therefore, to overcome this problem and increase the performance of the detectors, some researchers have used Machine learning (ML) techniques like LR (Logistic Regression), SVM (Support Vector Machine), DT (Decision Tree), RF (Random Forest), NB (Naive Bayes), and KNN (K-Nearest Neighbor) to detect types of code smell [7–9].

These ML-based (Machine Learning-based) techniques use historical datasets to learn about smelly and non-smelly items. Based on the learned knowledge, these techniques create a decision boundary or make rules to detect the code smells. The performance of the detectors depends on the quality of the training dataset. If the imbalanced dataset is used to train the ML techniques, the results may be biased towards the majority class (a class with more instances than others) [10]. In the case of the code smell problem, fewer smelly instances exist compared to non-smelly instances for training ML models. Due to that, the performance of the ML deteriorates in detecting smelly instances. To improve the performance, some researchers use oversampling imbalanced learning techniques. Where they generate synthetic instances for smelly items [11–13]. However, they use distance-based over-sampling

* Corresponding author.
E-mail addresses: 2021rcp9036@mnit.ac.in (P.S. Thakur), mahipaljadeja.cse@mnit.ac.in (M. Jadeja), sschouhan.cse@mnit.ac.in (S.S. Chouhan).

https://doi.org/10.1016/j.knosys.2024.111390
Received 9 September 2023; Received in revised form 30 December 2023; Accepted 8 January 2024
Available online 14 January 2024

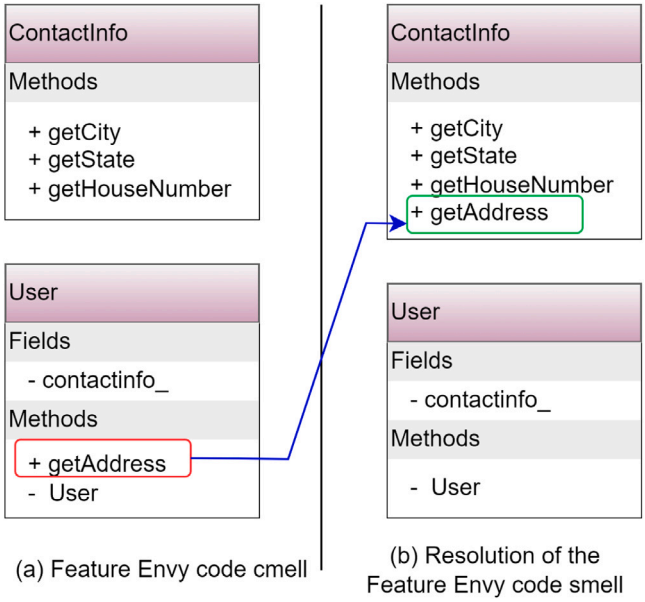

Fig. 1. Example to illustrate the Feature Envy code smell.

approaches that do not consider the inherent data distribution when generating synthetic instances.

To overcome this issue, in this work, we propose a novel oversampling technique, Cluster-Based Resampling Technique (*CBReT*), for imbalance learning in code smell problems. The *CBReT* technique is divided into four phases: Cluster Generation (**CG**), Synthetic Data Generation (**SDG**), Purifying Synthetic Instances (**PSI**), and Data Integration (**DI**). Where the CG formulates clusters from the dataset, and if the clusters are imbalanced, the SDG phase balances them by generating synthetic instances. Next, the *CBReT* employed a purifying synthetic instances (PSI) phase, which allows only valid synthetic instances (those that inherit more properties of the minority instances) to balance the dataset. In this way, the newly generated synthetic instance does not disturb the decision boundary between both classes. This advantage makes the CBReT technique more reliable and robust in addressing the data imbalance issue in overlapped code smell datasets. Subsequently, the DI phase combines all the generated instances with the original dataset to produce a balanced dataset.

The proposed technique has been investigated and evaluated on the four benchmark code smell datasets [14]. Each dataset represents one type of code smell such as (1) *Data Class*, when the class does not have all the appropriate methods; (2) *God Class*, when the class does too many tasks or contains many line codes; (3) *Feature Envy*: when a method is more interesting in another class functionalities, and (4) *Long Parameter*: when the method has a large number of parameters. The six machine-learning techniques are used to create the code smell prediction models. The prediction model is trained using the original and synthetic balanced dataset (generated by the *CBReT*), and then we compare their performance. The performance of the models are measured in terms of the f1-score, Matthews correlation coefficient (MCC), and Cohen's kappa. In summary, the major contribution of the proposed work is as follows

1. We present a novel oversampling technique, *CBReT*, to improve the performance of the code smell prediction model. It generates synthetic instances based on the inherent distribution of data.
2. *CBReT* has been evaluated on four publicly available benchmark code smell datasets. Extensive experiments have been conducted to evaluate the effectiveness of the *CBReT* technique.
3. The proposed technique has been compared with the other existing imbalanced learning techniques and state-of-the-art works. Statistical analysis also has been conducted to show the significant difference between the presented technique and the existing imbalance learning techniques.

To evaluate *CBReT*, we formulate the following research questions and pursuit to find the answers of these questions.

RQ1. Does the proposed technique (*CBReT*) improve the performance of the classifiers in detecting code smell?
RQ2. Does the proposed technique improve the ability of the classifiers to predict the smelly and non-smelly instances correctly?
RQ3. How does *CBReT* technique improve the performance of the classifiers over the state-of-the-art approaches?

The paper is organized as follows. Section 2 summarizes the related works. Section 3 describes the proposed *CBReT* technique. Section 4 presents the experimental setup and implementation details. Section 5 discusses the result analysis. The conclusions are given in Section 6.

2. Related work

After Martin Fowler coined the term code smell [15], numerous techniques have been proposed by researchers to detect it. Based on their detection methodology, these techniques can be categorized into manual and automatic approaches. The automatic approaches are also divided into tool-based [16–18] and machine-learning-based approaches [14,19]. Numerous work has been done to detect individual types of code smells using machine-learning techniques. Some of them are Maneerat et al. [20], Maiga et al. [21], and [22].

In this section, first, we focus on the oversampling imbalance learning approaches, and then we will focus on the use of imbalance learning in code smell detection.

2.1. Oversampling based imbalance learning techniques

Numerous researchers have used oversampling based imbalance learning techniques to synthesized the minority instances.

Nutthaporn Junsomboon et al. [23] proposed an imbalance learning approach to resolve the data imbalance issue. In this article, the authors combined the undersampling technique Neighbor Cleaning Rule (NCL) and the oversampling technique Synthetic Minority Over-Sampling Technique (SMOTE) to inherit the advantages of the under-sampling and over-sampling techniques. The NCL removes the data from the outlier in the majority class, and SMOTE generates the synthetic instances for the minority class. The model is tested on the medical datasets. Wenhao Zhang et al. [24] proposed combining the oversampling and ensemble approaches, WOT-Boost, to resolve the imbalance issues. This technique used the oversampling technique on the different phases of ensemble learning and improved the performance of the classifiers. The technique has been evaluated on the 18 publicly available datasets.

Junnan Li et al. [25] proposed an oversampling approach, NaNSMOTE, which extends the SMOTE. It used the natural neighbor and k nearest neighbor to generate the new synthetic instances. The NaNSMOTE generates instances near the center of the class, which provides more generalization of the synthetic instances. Mohammed H. IBRAHIM [26] proposed an approach outlier detection-based oversampling technique (ODBOT) to handle the multi-class imbalance problem. The ODBOT reduces the overlapping risk between the multiple classes while generating synthetic instances for minority classes. It first learns the dissimilarity relationships between the attributes and then uses minority instances to synthesize. Ashhadul Islam et al. [27] resolved the data imbalance issue in the small disjunct dataset or within the class. They proposed an oversampling approach, the K-Nearest Neighbor OveRsampling approach (KNNOR), which used three steps to find out the right areas for generating the synthetic instances. The approach used the overall distribution of the data to generate new instances. Xinmin Tao et al. [28] introduced an oversampling method that leverages

clustering to preserve the decision boundary of classes in overlapped datasets. Their proposed approach, named SVDDDPCO (a clustering-based oversampling technique), combines Support Vector Machines (SVM) to create hyperplanes with a focus on elevating the significance of minority instances. Additionally, it employs the Density Peaks Clustering (DPC) technique to cluster minority instances. The weights of the minority instances are then adjusted based on the cluster sizes. Specifically, the number of synthetic instances generated for each cluster is inversely proportional to its size.

In 2022, Wei Jianan et al. proposed imbalance learning techniques like Cluster-MWMOTE [29], Noise-Immunity Majority-Weighted Minority Oversampling Technique (NI-MWMOTE) [30], and Improving Adaptive Semi-Unsupervised Weighted Oversampling (IA-SUWO) [31] to resolve the data imbalance issues in the classification task. The Cluster-MWMOTE proposed to improve the performance of the fault diagnosis system. It applied the Agglomerative Hierarchical Clustering (AHC) technique to form the clusters of the minority instances. The authors applied the MWMOTE technique to balance the dataset on each identified cluster of the minority instances. They also used the MFO-LS-SVM technique to perform the classification task, optimizing the LS-SVM classifier's hyperparameter to improve diagnostic results. However, NI-MWMOTE first calculated the probability that the selected noise was real using the K-nearest neighbor technique. Next, the NI-MWMOTE formulated the clusters using the AHC technique and then calculated the number of required instances for each cluster. Subsequently, the NI-MWMOTE synthesized each cluster as per the requirements to balance the datasets. The IA-SUWO is the enhanced version of the adaptive semi-unsupervised weighted oversampling (A-SUWO) algorithm. It addresses the irregular distribution during the handling of data imbalance issues.

2.2. Imbalance learning techniques in code smell detection

In the real world, the instances of smelly items are very less than the non-smelly items. This imbalance affects the performance of the learning algorithms. To address this issue many researchers proposed or used imbalance learning techniques.

F Pecorelli et al. [12] used five balancing techniques to improve the performance of the machine learning techniques. In the result analysis, the authors suggested that the synthetic data generated by the SMOTE improved the classifiers' performance. Manuel De Stefano et al. [32] proposed a model based on the cross-project machine learning approaches. This model used the instances of the different projects to balance the dataset. It included the smelly items from the different projects to increase the instances of the minority class. The object of this model was to provide the necessary information about the smell items to the classifier to learn more about the code smell. However, it did not impact the performance of the classifiers. Yang Zhang et al. [11] again proposed a model MARS to detect brain class code smell. It used the residual networks to improve the gradient degradation. It introduces a metric-attention mechanism, which increases the weight of the essential metrics to label smelly instances. To balance the dataset, it also uses the SMOTE (Synthetic Minority Over-sampling Technique) synthesizer.

Yang Zhang et al. [33] proposed a model DelSmell, to detect the code smell like Brain class and Brain Method using deep learning models. In order to build the dataset, 24 real-world projects have been used to extract the samples. It also solves the problem of imbalanced learning. In order to balance the dataset, it replaces the non-smelly code of lines with the smelly code of lines, i.e., the calling of the method is replaced by its definition. Seema Dewangan et al. [13] proposed a technique to detect code smells. In this article, the author used two deep learning and five ensemble learning technique to classify the code smells. The model has been tested on the four code smells i.e., Long method, Data Class, God Class, and Feature Envy. The proposed technique also handles the imbalanced data issue by balancing the dataset.

Table 1
Variables and symbols used.

Symbols	Discription
$\mathcal{D}$	Imbalanced Code Smell dataset
$\mathcal{D}'$	Balanced Code Smell dataset
$\mathcal{N}$	Total number of instances in datasets
n	Number of the features in the Code Smell dataset ($\mathcal{F}_1, \ldots \mathcal{F}_n$)
$\mathcal{L}_1$	Class of the smelly samples
$\mathcal{L}_0$	Class of the non-smelly samples
l	It is an instance of dataset ($l \in \mathcal{D}$)
l'	Synthetic instance near to the l
$\mathcal{F}_i^{\mathcal{L}_1}$	The ith feature of the smelly class
$\mathcal{F}_i^{\mathcal{L}'_1}$	The ith feature of the synthetic smelly instance
c_i	The ith cluster out of κ clusters
$c_{i_{min}}$	Minority instances of the selected ith cluster
$c_{i_{maj}}$	Majority instances of the selected ith cluster
μ_{c_i}	Mean of the ith cluster
α	Significance level in statistical test
ρ_{c_i}	Variance of the ith cluster
ω_{c_i}	Weight of the instance with respect to the ith cluster
κ	Number of the clusters
rd	Random number between 0 and 1

To balance the dataset, the model used SMOTE synthesizer. Jatin Nanda et al. [34] proposed a model SSHM (Smote-Stacked Hybrid Model) to improve the performance of the Machine learning algorithms in terms of code smell prediction. This model combined the SMOTE and Stacking to balance the dataset. The four datasets, Feature envy, God Class, Long Method, and Data Class, have been used to evaluate the model. Sofien Boutaib et al. [35] proposed ADIODE, an effective search-based technique to detect and identify the code smell in an imbalanced environment. The ADIODE is based on the Evolutionary Algorithm, which uses f-measure as a fitness function. The ADIODE was evaluated on the seven code smell datasets.

In this section, we discussed resampling techniques, including oversampling and undersampling, commonly used to address data imbalance in various domains. Prior research has demonstrated the effectiveness of resampling methods in dealing with data imbalance. However, code smell datasets typically have a limited number of instances, making undersampling impractical. Moreover, these datasets often feature overlapping instances between the minority and majority classes, making the direct application of oversampling challenging. In this paper, we introduce CBReT, an oversampling approach that leverages data distribution by creating clusters to synthesize datasets.

3. Proposed work

This section presents the proposed *CBReT* technique to generate synthetic data. The formulation of the imbalance problem in the code smell is given in Definition 1. The CBReT technique follows an oversampling approach. To begin, *CBReT* formulates clusters based on the data distribution using the Gaussian Mixture Model (GMM). Each cluster represents a sub-dataset containing both minority and majority instances. *CBReT* processes one cluster at a time. It determines the number of additional instances required to balance the sub-dataset by subtracting the number of minority instances from the majority instances. Subsequently, synthetic instances are generated using the minority instances, and these synthetic instances are combined with the original dataset. This process is repeated for each cluster, excluding clusters that are already balanced. Finally, all the clusters are concatenated into a single, balanced dataset. The *CBReT* technique can be divided into four phases: Cluster Generation (**CG**) (Section 3.1), Synthetic Data Generation (**SDG**) (Section 3.2), Purifying Synthetic Instances (**PSI**) (Section 3.3), and Data Integration (**DI**) (Section 3.4), as shown in Fig. 2. The following subsections provide a detailed description of each of these four phases. The description of the variables and symbols that have been used is given in Table 1.

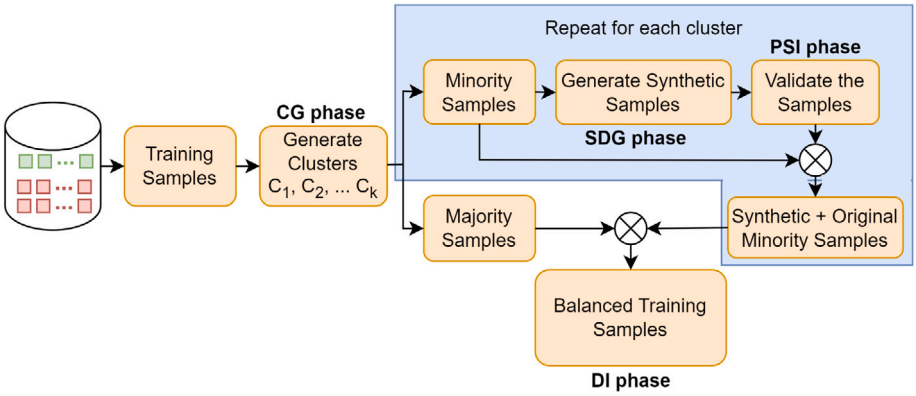

Fig. 2. Proposed methodology to generate synthetic instances for Code Smell prediction model.

Definition 1 (*Imbalanced Dataset [36]*)**.** A code smell dataset D_{CS} containing n features $\mathcal{F}_1, \ldots, \mathcal{F}_n$ and $\mathcal{N}$ number of instances is called imbalanced if $|\mathcal{L}_1| \ll |\mathcal{L}_0|$ or vice versa.

Where $\ll$ represents the significantly less in numbers, $|\mathcal{L}_1|$ is the count of smelly instances and $|\mathcal{L}_0|$ is the count of the non-smelly instances.

Objective: The objectives of the CBReT technique are:

1. To generate balanced synthetic code smell dataset $\mathcal{D}'$ containing equal number of smelly and non-smelly items, Here, $\mathcal{D}' = (\mathcal{L}_1 \cup \mathcal{L}'_1) \cup \mathcal{L}_0$ where $|\mathcal{L}_1 \cup \mathcal{L}'_1| \approx |\mathcal{L}_0|$.
2. To generate the synthetic instances in such a way that the mutual properties between any pair of the features $\mathcal{F}_i^{\mathcal{L}_1}$ and $\mathcal{F}_i^{\mathcal{L}'_1}$ remain similar.

3.1. Cluster generation (CG)

This phase learns the inherent distribution of the code smell data. The CG takes a training dataset D as an input, which has both minority and majority instances. Next, using the elbow method [37], the *CBReT* calculates the suitable value of the κ, where κ represents the optimal number of clusters $(c_1, \ldots, c_\kappa)$. Finally, it uses Gaussian Mixture Models (GMM) to create clusters based on the data distribution. The GMM does not segregate clusters using the distance from the centroid. It estimates the parameters like mean (μ), variance (ρ), and weight (ω) of the clusters. These parameters are used to find the probability of the given data point's membership value for every cluster. Data points are clustered based on their highest probability. After adding every new instance in a cluster, the values of the parameters (μ, ρ, and ω) for the cluster are updated.

Suppose we have a dataset (X) containing κ clusters and mean, variance, and weight of the c_i^{th} cluster are μ_{c_i}, ρ_{c_i}, and ω_{c_i} respectively. If we have m_{c_i} number of the data points in c_i^{th} cluster and θ is the set of the parameters (μ_{c_i}, ω_{c_i}, ρ_{c_i}) for the each c_i^{th} component, then a likelihood estimation is calculated using Eq. (1):

$$l(\theta|X_{c_i}) = \sum_{j=1}^{m_{c_i}} \log p(x_j|X_{c_i};\ \mu_{c_i}, \rho_{c_i}) + \log p(X_{c_i};\ \omega_{c_i}) \tag{1}$$

The goal of maximum likelihood estimation is to find the values of the model parameters that maximize the likelihood function over the parameter space, shown in Eq. (2):

$$\theta' = \arg\max_{\theta} l(\theta|X_{c_i}) \tag{2}$$

Next, to maximize the likelihood function for the c_i^{th} cluster, GMM calculates the partial derivative over the μ_{c_i}, ω_{c_i}, ρ_{c_i}.

$$\mu_{c_i} = \frac{\sum_{j=1}^{m_{c_i}} \{X_j = c_i\} x_j}{\sum_{j=1}^{m_{c_i}} \{X_j = c_i\}} \tag{3}$$

$$\rho_{c_i} = \frac{\sum_{j=1}^{m_{c_i}} \{X_j = c_i\}(x_j - \mu_{c_i})(x_j - \mu_{c_i})^T}{\sum_{j=1}^{m_{c_i}} \{X_j = c_i\}} \tag{4}$$

$$\omega_{c_i} = \frac{1}{m_{c_i}} \sum_{j=1}^{m_{c_i}} \{X_j = c_i\} \tag{5}$$

Eqs. (3), (4), and (5) are used to update the mean, variance, and weight parameters, respectively.

Algorithm 1: *GetClusters*(X, κ)

```
Input: X, κ, μ_{c_i}, ρ_{c_i}, and ω_{c_i}
Result: X_1, ..., X_κ
Initialize μ_{c_i}, ρ_{c_i}, and ω_{c_i} by some random values where
  1 ≤ c_i ≤ κ.
while not converge do
    X_{c_i} ← P(c_i|X)        ▷ Probability of X with respect to cluster c_i
    μ_{c_i} ← update(μ_{c_i}, X_{c_i})        ▷ Using Equation (3)
    ρ_{c_i} ← update(ρ_{c_i}, X_{c_i})        ▷ Using Equation (4)
    ω_{c_i} ← update(ω_{c_i}, X_{c_i})        ▷ Using Equation (5)
end
return X_1, ..., X_κ
```

Here, the Algorithm 1 iterates until it converges. Convergence is determined by the *tol* parameter, which controls the relative tolerance concerning the change in log-likelihood. The value of *tol* is set to 0.001. Thus, if the log-likelihood value becomes less than *tol* in two consecutive steps, the algorithm is considered to have converged. In case the algorithm fails to maintain the likelihood below the *tol*, we forcibly stop it when the number of iterations reaches $max\_itr = 100$ and return the best estimate it has found up to that point. Both *max_itr* and *tol* are the hyperparameters and can be tuned as per the datasets' behavior.

Algorithm 1 shows the process to generate the κ clusters of the input data (training dataset). Each cluster may contain both (minority and majority) types of instances. Further, each cluster is treated as an imbalanced dataset. Now, these clusters go into the SDG phase, which generates the synthetic instances with respect to each cluster.

3.2. Synthetic data generation

It generates synthetic instances of the minority class. In Algorithm 2, lines from 4 to 13 represent the SDG phase. Here, *CBReT* algorithm first calculates the required number of synthetic instances by calculating the difference between the count of the majority and minority instances for the cluster. Next, it randomly selects two instances (l_{r1} and l_{r2}) within

a cluster from the minority class and calculates the new instance by using Eq. (6).

$$l'_{r1} = l_{r1} + rd \times (l_{r2} - l_{r1}) \tag{6}$$

where rd is an generated random number between 0 and 1.

Algorithm 2: *CBReT* Technique

```
   Data: D(X_train, y_train)
   Result: D'
1  newData=[ ]
2  c_1, ..., c_κ ← GetClusters(D, κ)
3  for each c_i ∈ [c_1, ..., c_κ] do
4      μ_{c_i} ← GetMean(c_i)
5      count ← 0
6      Segregate the minority and majority class from the dataset c_i and store them
         in c_{i_min} and c_{i_maj}, respectively.
7      req_instances ← ||c_{i_min}|| − ||c_{i_maj}||
8      while count ≠ req_instances do
9          L_{r1} ← SelectRandom(c_{i_min})
10         L_{r2} ← SelectRandom(c_{i_min})
11         if L_{r1} ≠ L_{r2} then
12             rd ← randomNumber(0,1)
13             l'_{r1} ← l_{r1} + rd × (l_{r2} − l_{r1})
14             d_1 ← d(μ_{c_i}, L'_{r1})
15             d_2 ← d(μ_{c_i}, L_{r1})
16             d_3 ← d(μ_{c_i}, L_{r2})
17             if d_2 > d_3 then
18                 if d_3 < d_1 < d_2 then
19                     newData ← newData ∪ L'_{r1}
20                     count ← count + 1
21                 end
22             else
23                 if d_3 > d_1 > d_2 then
24                     newData ← newData ∪ L'_{r1}
25                     count ← count + 1
26                 end
27             end
28         end
29     end
30 end
31 D' ← D ∪ newData
```

Here, rd is used to provide randomness. The advantage of rd is that whenever the algorithm chooses the previously used pair of data points, the random number provides a slight deviation. This way, the *CBReT* can generate multiple instances using the same pair of the original data instances. Similarly, randomly choosing the two data instances to generate new synthetic data provides randomness in the data distribution. In a cluster, all the minority instances have similar properties; therefore, unlike SMOTE, there is no need to calculate the nearest neighbors. The instances of the clusters have already been chosen based on the data distribution. Therefore, the random selection of the two instances provides a fair chance to all minority instances in a dataset for generating synthetic instances. Since the selected data belong to the same cluster, they represent similar properties, and the generated synthetic instance also inherits the properties of the cluster. In each iteration, the *CBReT* generates one synthetic instance and checks the validity of the generated instance in the PSI phase.

3.3. Purifying synthetic instances (PSI)

It uses the mean μ_{C_i} of the dataset (input cluster) to validate the synthetic instance. In Algorithm 2, lines from 14 to 28 represent the PSI phase. Initially, the PSI phase calculates the Manhattan distance between the μ_{C_i} and both selected instances using Eq. (7).

$$d(p,q) = \sum_{i=1}^{n} |\mathcal{F}_{pi} - \mathcal{F}_{qi}| \tag{7}$$

where p and q are the two data points (instances) and $\mathcal{F}_{pi}$ and $\mathcal{F}_{qi}$ are their ith features, respectively.

Suppose, the distance of the selected instances l_{r1} and l_{r2} from the μ_{c_i} are $d(\mu_{c_i}, l_{r1})$ (d_2) and $d(\mu_{c_i}, l_{r2})$ (d_3), respectively. PSI also calculates the distance between the μ_{c_i} and l'_{r1} which has denoted as $d(\mu_{c_i}, l'_{r1})$ (d_1). Further, PSI checks the position of the generated instance based on the distance from the mean. For that, if $d_2 > d_3$ then, $d_2 > d_1 > d_3$. Similarly, if $d_2 < d_3$ then, $d_2 < d_1 < d_3$. The validation process is also shown in the Algorithm 2 at lines 15 to 28. The algorithm keeps the synthetic instance if it passes the validation condition; if it fails, then the instance will be rejected. After the first iteration process, we will go back to the SGD phase for the next synthetic instance. The SDG and PSI phases execute sequentially in each iteration until all required instances are not generated. This process will be repeated for each cluster. Next, to combine the generated instances with the original dataset, *CBReT* follows the DI phase.

3.4. Data integration (DI)

It is a phase where *CBReT* combines the actual and synthetic instances to generate a balanced dataset D'. For this, DI receives the list of the required valid instances from the SDG and PSI phases. Next, *CBReT* integrates the received valid instances list with the actual dataset to get a balanced dataset (Algorithm 2, line 31).

Next, the generated D' dataset is used to train the code smell prediction model. Whereas the model predicts the code smells by using the training knowledge. Based on the prediction, the performance measures are recorded, which will be used to analyze the performance of the code smell prediction model.

Next, let us understand the synthetic data generation process using a simple example.

3.5. Synthetic instance generation: Example

While generating the instances, there can be two cases: the generated instance will be valid or invalid. Let us take a generalized example that shows how the *CBReT* technique generates valid synthetic instances. Suppose the dataset (cluster) has (1,6), (2,3), (4,5), and (5,1) four minority instances and (3,1), (5,2), (1,4), (4,3), (4,6), and (3,4) six majority instances (shown in Fig. 3). Here, the black dot represents the majority instances; the blue triangle represents the actual minority instances; the yellow square shows the centroid of the dataset; the green and red triangles symbolize valid and invalid synthetic instances, respectively. The dotted lines show the distance between the actual instance and the mean. Similarly, the dashed lines show the distance between the synthetic instance and the mean. Fig. 3(a) and (b) show the illustration of valid and invalid synthetic instances, respectively.

1. *Case 1 (Valid Case):* First, *CBReT* calculates the mean of the given dataset (cluster) using Eq. (3), which is $\mu_{c_i} = (3.2, 3.5)$, then it selects two instances ($\mathcal{L}_{r1}$ and $\mathcal{L}_{r2}$) randomly from the minority class. Assume that the selected instances $\mathcal{L}_{r1}$ and $\mathcal{L}_{r2}$ are (5,1) and (2,3), respectively. Now, by using Eq. (7), it calculates the values of d_2 and d_3 which are 4.3 and 1.7, respectively. Next, the *CBReT* technique uses Eq. (6) to generate a synthetic instance, where suppose $rd = 0.15$.
   $\mathcal{L}'_{r1} = (5,1) + 0.15 \times ((2,3) - (5,1))$
   $\mathcal{L}'_{r1} = (4.5, 1.3)$
   After the calculation, the new synthetic instance $\mathcal{L}'_{r1} = (4.5, 1.3)$ is generated. Further, the validation phase will start. Here, the *CBReT* calculates the d_1, which is 3.52. Next, it follows steps 15 to 25 from Algorithm 1 to check whether the $\mathcal{L}'_{r1}$ is valid or not. Here, we can see that the condition $d_3 < d_1 < d_2$ is true. This means the $\mathcal{L}'_{r1}$ is the valid instance, as shown in Fig. 3(a).
2. *Case 2 (Invalid Case):* Let us assume that in the next iteration, the selected instances $\mathcal{L}_{r1}$ and $\mathcal{L}_{r2}$ are (5,1) and (4,5), respectively, and the calculated Manhattan distance of both points from the mean are d_2= 4.3 and d_3= 2.3. Thus, using the following selected instances, the *CBReT* generates the synthetic instance by Eq. (6), with the random number $rd = 0.73$.

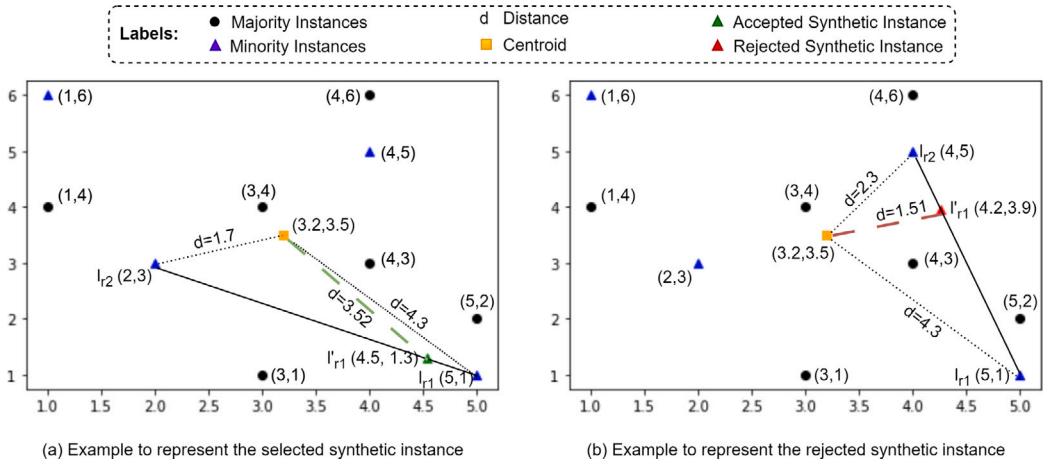

Fig. 3. Example to illustrate how to check the validity of the generated instances.

$\mathcal{L}'_{r1} = (5,1) + 0.73 \times ((4,5) - (5,1))$
$\mathcal{L}'_{r1} = (4.2, 3.9)$
The above calculation shows that the generated synthetic instance is $\mathcal{L}'_{r1} = (4.2, 3.9)$. Also, the calculated distance using Eq. (7) of the $\mathcal{L}'_{r1}$ from the mean is $d_1 = 1.51$. Next, the PSI phase checks the validity of the generated instance by following steps 15 to 25 from Algorithm 2. Here, we can see that $\mathcal{L}'_{r1}$ did not satisfy the specified condition. Where the d_1 does not lie between the d_2 and d_3. Therefore the generated instance will be rejected, as shown in Fig. 3(b).

Based on the above example, $\mathcal{L}'_{r1}$ is more likely to be accepted if it originates near $\mathcal{L}_{r1}$ and $\mathcal{L}_{r1}$ is further away from the mean than $\mathcal{L}_{r2}$.

Filtering the instances provides the advantage of making synthetic instances more realistic and effective compared to other oversampling approaches like SMOTE. SMOTE selects random points from the dataset and utilizes their neighbors to generate synthetic instances. Since the selected point and its neighbors can be anywhere in the dataset, the position of the synthetic instances cannot be controlled. In cases where the dataset is highly overlapped, the minority and majority instances can be very close to each other. If we were to include all synthetic instances without validation, some of these instances might disrupt the class-separating boundary (hyperplane), potentially leading to a negative impact on classifier learning. The PSI phase ensures that only instances with properties similar to the minority instances are selected.

4. Experimental setup

This section describes the datasets (Which are considered for the experiments), performance measures, machine learning techniques, and implementation details to set up the experiments.

4.1. Code smell dataset description

We have used the publicly available four benchmark code smell datasets provided by Fontana et al. [14] that are used in previous studies [9,13,38–40]. These datasets are the most frequently used in the previous work. Apart from these datasets, some researchers have created their own datasets but have not provided publicly. The datasets employed in this study were collected from seventy-four software systems.[1] Five code smell detection tools like iPlasma, Fluid Tool, Marinescu detection rule, Anti-Pattern Scanner, and PMW are used to detect the basic code smells like Data Class, God Class, Long Method, Feature Envy, Switch Statement, and Long Parameter. The final generated data has been manually verified based on the definition of the code smells by the three graduate students. Fontana et al. used only four out of the six datasets in the experiments. The code smells have been selected based on their impacts on the qualities of the software. From these available datasets, we have used three datasets: Data Class, Feature Envy, and Long Parameter. Apart from these, another used dataset is God Class dataset[2] which is provided by Aleksandar Kovačević et al. [41]. Here, we did not directly apply these datasets in our experiments. Since our work is in imbalance learning, therefore if the original dataset provided by the Fontana et al. [14] and Aleksandar Kovačević et al. [41] is balanced or near to the balanced we modified it. In these case, first, we remove some instances from the smelly class to increase the imbalance ratio between the smelly and non-smell classes and then employ the resulting dataset in the experiments. The descriptions of the modified datasets are given in Table 2. The datasets of Data Class, God Class, Feature Envy, and Long Parameter are represented as Dataset 1, Dataset 2, Dataset 3, and Dataset 4, respectively.

Table 2
Description of the code smell datasets.

Dataset	Metrics	Non-Smelly	Smelly	Total	Imbalance ratio
Data class	63	183	94	277	1.90
God class	46	2049	251	2300	8.16
Feature envy	82	280	140	420	2.00
long Parameter	55	282	69	351	4.08

4.2. Experimental procedure

The proposed code smell prediction model uses four code smell datasets, six machine learning techniques, and the *CBReT* (proposed technique) to perform the binary classification task. Each dataset represents one type of code smell containing smelly and non-smelly items.

We performed experiments for each dataset to evaluate and compare the CBReT technique with state-of-the-art approaches. In the experimental setup, we initially divided the entire dataset into training and testing sets following an 80 : 20 ratio, utilizing the stratified technique. This stratified approach ensured a balanced ratio between the smelly and non-smelly classes in both the training and testing sets. Next, we applied various imbalance learning techniques, including the CBReT technique, to balance the training dataset. For fairness and consistency, we created multiple copies of the training set, allocating

[1] https://github.com/hjamaan/IST2021-CodeSmellStackingEnsemble/tree/main/Datasets/Original

[2] https://codeocean.com/capsule/5256791/tree/v1l

Table 3
Description of the used performance measures.

Performance measures	Description
Precision	It represents how much the predicted model is precise for smelly and non-smelly items. $precision = \frac{TSI}{TSI+FSI}$
Recall	Recall measures the validity of the prediction model, where it checks how many smelly items are predicted correctly from the total tested smelly items. $recall = \frac{TSI}{TSI+FNSI}$
F1-Score	The F1-score is the harmonic mean of the precision and recall. $f1-score = \frac{2 \times precision \times recall}{precision+recall}$
Matthews Correlation Coefficient (MCC)	The MCC provides the correlation between the actual and predicted values. Davido Chicco [45] says that MCC performs better if the prediction model works fine for both (positive and negative) classes. In our case, we want to check whether the model correctly predicts the smelly or non-smelly items, that is why we included MCC measures. $mcc = \frac{(TSI \times TNSI)-(FSI \times FNSI)}{\sqrt{(TSI+FSI)(TNSI+FSI)(TSI+FNSI)(TNSI+FNSI)}}$
Cohen's Kappa	In terms of the imbalanced dataset, it provides more realistic information about the performance of the model compared to the accuracy. Cohen's Kappa measure is estimated by taking the imbalance distribution into account. $cohen's\ kappa = \frac{ac_0-ac_e}{1-ac_e}$ Where ac_0 is the overall model's accuracy and ac_e is the sum of the probability that the prediction agrees with actual values by chance for the non-smelly class and for the smelly class, respectively.

one copy for each imbalance learning technique. Subsequently, each copy of the training set was balanced using the designated imbalance learning technique. The balanced training set was then utilized to train a code smell detection model for each technique. This process was repeated ten times, and the mean performance measures were reported for each imbalance learning technique using the original testing set.

We also employed the ten-fold cross-validation technique for each imbalance learning technique. The results have been included in Appendix, specifically in Tables A.12, A.13, and A.14. Upon comparing the results obtained from both techniques, namely the ten-fold cross-validation technique and the ten-time repeating process, we did not observe any significant differences in essential performance measures such as F1-score, MCC, Cohen's Kappa, etc.

4.3. Performance measures

To evaluate the code smell prediction model, we used performance measures like precision, recall, f1-score, cohen's kappa, and MCC. We have excluded the accuracy because, in terms of the imbalanced dataset, accuracy might be unreliable [42]. Consider this example, suppose we have a prediction model which can correctly predict all the non-smelly items but fails to predict a single smelly item correctly. If we use this model to predict smelly items on a dataset with 95 non-smell items and five smelly items, then according to the model's nature, its accuracy would be 95%, which is excellent accuracy. However, the model wrongly predicts all the smelly items, therefore if we see in terms of the smelly class, the model's accuracy is 0%. Therefore we exclude the accuracy from the measures. Some previous studies say that the f1-score, kohen's kappa, and MCC play a vital role in measuring performance on imbalanced datasets [43,44], therefore we include them.

Let us consider the confusion matrix for the code smell prediction model, which has the following parameters: TSI (True Smelly Items), TNSI (True Non-smelly Items), FSI (False Smelly Items), and FNSI (False Non-smelly Items). Based on there parameters, the description of the performance measures are given in Table 3.

Further, we have performed statistical analysis using With Wilcoxon signed-rank test. The details are given below.

Wilcoxon signed-rank test: It is a non-parametric statistical analysis test [46]. Wilcoxon signed-rank test is used to compare two samples. Here, we used it to show the significant difference between the results of the two methods. It formulates the hypothesis (Null and alternative). Next, it finds the evidence for the formulated hypothesis. For this, the Wilcoxon signed-rank test first calculates the difference between each pair of both samples. After that, based on the difference score value, it calculates the sign rank (positive and negative), means, and standard deviation. Now, it calculates the p-value and r-value using the mean and standard deviation. If the p-value is less than the critical value ($\alpha = 0.05$) then it says the null hypothesis will be rejected, and the alternative will be accepted. In this experiment, we assume two null hypotheses: first between the imbalanced and synthetic balanced dataset, and second between the results of the *CBReT* and other existing imbalance learning techniques. The formulated null hypothesis for the imbalance and synthetic balanced dataset is:

- *Null hypothesis (H_0):* There is no significant difference in the performance of the prediction model trained by the original (imbalanced) and synthetic balanced dataset.

Similarly, the null hypothesis between the results of the *CBReT* and imbalance learning techniques is:

- *Null hypothesis (H_0):* There is no significant difference in the performance of the prediction model trained by the generated resampled dataset using the *CBReT* and other imbalance learning techniques.

4.4. Machine learning techniques used to build prediction model

The six machine-learning techniques have been used to build a model for code smell prediction. The previous work shows that these techniques perform well for code smell detection [9,47,48]. We have used these six machine-learning techniques to evaluate and compare the proposed work with other state-of-the-art. The techniques used include Random Forest (RF), Logistic Regression (LR), Naïve Bayes (NB), K-nearest neighbor (KNN), Support Vector Machine (SVM), and decision tree (DT).

DT is a tree-based classifier that constructs a tree-type structure to represent data information and classify modules [49]. RF, on the other hand, constructs multiple decision trees from random samples [50]. Logistic regression estimates the relationship between a dependent variable and multiple independent variables, using a sigmoid function to classify the modules [51]. Naive Bayes employs the Bayes theorem to calculate the probability of the occurrence of the dependent variable

and assumes that all variables are independent of each other [52]. SVM finds the optimal hyperplane to separate data into different classes by maximizing the margin between the classes [53]. KNN is a lazy learner that determines neighbors of the given modules using a distance measure and classifies the given module based on the majority voting of the neighbors [54].

4.5. Implementation details

All the experiments have been done in Python language. Firstly, the datasets divided into 80 : 20 ratios using the stratified technique in training and testing sets. The proposed technique is used to balance the training dataset only, where it creates clusters based on the data distribution. Each cluster is used as a separate imbalance dataset, which is the subset of the original dataset. To get the required synthetic instances for the balancing, *CBReT* calculates the difference between the number of minority and majority instances. Here, *CBReT* has used the SDG and PSI phases (discussed in Sections 3.2 and 3.3) to generate valid synthetic instances for the minority class. After balancing the dataset, we have used six machine-learning techniques (discussed in Section 4.4) for the classification. The performance of the classifiers is tested by the testing dataset, which has already been separated from the dataset. For fairness, all the prediction models, whether it is trained using the balanced or not-balanced dataset, have been tested by the same testing dataset.

Mean of the measures: The original and synthetic balanced dataset is used for training the detection model. The above process has been repeated ten times to get more precise results. After completing the experiments, we calculate the mean for each measure.

We also employed the ten-fold cross-validation technique for each imbalance learning technique. The results have been included in Appendix, specifically in Tables A.12, A.13, and A.14.

5. Results and discussion

In this section, we use the experimental setup and proposed technique discussed in Sections 4 and 3 to evaluate the code smell prediction model. For the sake of the validation, the experiment has been performed ten times for each dataset, and the mean values of the performance measures have been calculated to get the results. We compare our model, *CBReT*, with other baseline models (models using the imbalanced dataset and models employing other state-of-the-art techniques). Both *CBReT* and the other state-of-the-art approaches utilize the same training samples to generate synthetic instances for the minority class.

5.1. RQ1: Does the proposed technique (CBReT) improve the performance of the classifiers in detecting code smell?

To address this research question (RQ), we create two prediction models: one using the imbalanced dataset and another using the balanced dataset generated by the *CBReT* technique. Both models employ six fundamental machine-learning techniques discussed in Section 4.4 for prediction. For training the prediction models, we utilize the original (imbalanced) dataset in the first model, referred to as the "Imb" model. In the second model, known as the "Bal" model, we employ the synthetic balanced dataset generated by the *CBReT* technique. Tables 4 and 5 display the results, with Table 5 summarizing the outcomes using labels such as"W" (Win), "L" (Loss), and "D" (Draw). We calculate the mean difference between the performance measures of the Imb and Bal models. A "loss" occurs when the Bal model performs worse than the Imb model, a "draw" when they perform equally, and a "win" when the Bal model outperforms the Imb model.

From Tables 4 and 5, we can observe that the *CBReT* technique improved the performance of all ML techniques in terms of all essential measures. In Table 4, we can see that precision has outperformed for Datasets 1 and 4, reaching the highest value of 0.92 and the lowest value of 0.59 when employing *CBReT*. Simultaneously, the recall has shown improvement across all four datasets, with the highest value of 0.95 observed for Dataset 3 and the lowest value of 0.58 recorded for Dataset 1. To assess the overall performance of the classifiers, we considered the F1-score measure across all datasets. The F1-score shows improvement for all datasets. For Dataset 3, *CBReT* reported the highest F1-score of 0.94 with the DT technique, while the lowest value of 0.53 was observed for Dataset 4 with the SVM classifier. The F1-score takes into account both precision and recall values. Therefore, we can conclude that *CBReT* has enhanced the performance of the base classifiers through dataset resampling. Next, the G-mean and AUC performance measures also demonstrated improvements across all datasets. The highest G-mean value, 0.95, was observed for Dataset 3, while the lowest value of 0.60 was recorded for Dataset 1. Likewise, for the AUC, the highest value was 0.87, recorded for Dataset 1, and the lowest was 0.19, recorded for Dataset 2. To assess the overall improvement in all measures, we calculate the mean difference between the results of the imbalanced and synthetic balanced models for all datasets, the results are reported in Table 5. Notably, recall, f1-score, Cohen's kappa, G-Mean, and AUC outperform in all four datasets across six baseline learning models. However, MCC shows decrement in one comparison (in the SVM model for Dataset 2), but it shows an improvement in all other presented scenarios. Here, all the learning techniques show 100% wins for Recall, F1-score, MCC, and Cohen's Kappa except SVM. SVM shows a 100% win for the recall and f1-score; however, it shows a 75% win for MCC. The precision shows a 50% win for all the learning techniques except NB; for NB, it shows a 75% win.

Further, to find additional evidence and show significant differences between the results of both models (with imbalanced and synthetic balanced datasets), we perform statistical analysis, which is at the bottom of Table 5. The p-value shows that there is a statistically significant difference between the performance of the prediction model with an imbalanced dataset and with the synthetic balanced dataset. From the statistical result, we can clearly see that the synthetic balanced dataset improves the performance of the learning techniques. To visualize the distribution of both (imbalanced and synthetic balanced) datasets, we plot a graph using the PCA for all the four datasets, shown in Fig. 4. In this figure, we can see that the newly generated synthetic instances are very close to the minority instances, which helps the learning techniques during the training to distinguish between the minority and majority instances by fair learning.

Based on the above observations, we can conclude that the synthetic data generated by the *CBReT* technique improved the performance of all classifiers with respect to most of the performance measures. In the end, we can summarize that based on the performance difference and the above observations, the performance of all six machine-learning techniques have improved, but the LR, SVM, and KNN show the most improvements compared to DT, RF, and NB.

5.2. RQ2: Does the proposed technique improve the ability of the classifiers to predict the smelly and non-smelly instances correctly?

To find the answer to this research question, we have performed the experiment using the original imbalanced and synthetic balanced (using the *CBReT* technique) dataset and calculated the MCC measures. The MCC measure is calculated by considering all four parameters of the confusion matrics (TSI, TNSI, FSI, and FNSI). Aa per Davide Chicco [45], the MCC measure increases when the prediction model predicts both (positive and negative) instances correctly. Therefore, we calculate the MCC value for all the datasets to check how much our proposed technique improved the prediction ability of the learning techniques. Table 6 shows the difference between the MCC value of the learning techniques using an original imbalanced and balanced (using the *CBReT* technique) dataset. Here, the positive value shows how much better results we can get using the *CBReT* balanced dataset,

Table 4
Comparison Results between imbalanced dataset and synthetic balanced dataset using the different learning classifiers.

Classifier	Precision		Recall		F1-score		Cohen_Kappa		MCC		G-Mean		AUC	
	Imb	Bal	Imb	Bal	Imb	Bal	Imb	Bal	Imb	Bal	Imb	Bal	Imb	Bal
Dataset 1														
Logistic Regression	0.6715	0.7388	0.6708	0.7355	0.6614	**0.7364**	0.3326	0.4733	0.3423	0.4743	0.6791	0.7421	0.6813	0.7216
Support Vector Machines	0.6858	0.6923	0.5469	0.5867	0.4236	**0.5578**	0.0862	0.1715	0.1747	0.2398	0.5426	0.6210	0.5970	0.6406
Decision Trees	0.8460	0.8579	0.8475	0.8519	0.8450	**0.8541**	0.6907	0.7084	0.6935	0.7097	0.8445	0.8551	0.8234	0.8324
Random Forest	0.8574	0.9150	0.8622	0.9110	0.8576	**0.9127**	0.7160	0.8254	0.7196	0.8260	0.8608	0.9083	0.8478	0.8777
Naive Bayes	0.7037	0.7255	0.6665	0.6771	0.6286	**0.6349**	0.3121	0.3306	0.3683	0.3997	0.6751	0.6857	0.6991	0.7083
K-Nearest Neighbor	0.5490	0.5927	0.5386	0.5932	0.4957	**0.5865**	0.0721	0.1819	0.0870	0.1859	0.5408	0.6009	0.5824	0.6188
Dataset 2														
Logistic Regression	0.7727	0.6830	0.5808	0.7429	0.6087	**0.7052**	0.2361	0.4128	0.2955	0.4212	0.5910	0.7324	0.2141	0.2788
Support Vector Machines	0.7187	0.6342	0.6885	0.7426	0.6491	**0.6569**	0.3232	0.3282	0.3747	0.3607	0.6168	0.7073	0.2071	0.2627
Decision Trees	0.6398	0.6362	0.6073	0.6604	0.6191	**0.6453**	0.2404	0.2923	0.2444	0.2951	0.6073	0.6534	0.1880	0.1903
Random Forest	0.8049	0.6680	0.5607	0.7609	0.5806	**0.6955**	0.1883	0.3982	0.2698	0.4183	0.5575	0.7499	0.1782	0.2744
Naive Bayes	0.7358	0.7155	0.7247	0.7589	0.7295	**0.7340**	0.4592	0.4688	0.4600	0.4723	0.7235	0.7773	0.3161	0.3448
K-Nearest Neighbor	0.8047	0.6809	0.5880	0.8010	0.6201	**0.7130**	0.2588	0.4360	0.3263	0.4663	0.5917	0.7971	0.2172	0.3175
Dataset 3														
Logistic Regression	0.7406	0.7387	0.7165	0.7712	0.7263	**0.7493**	0.4538	0.5016	0.4565	0.5089	0.7170	0.7712	0.4689	0.5047
Support Vector Machines	0.7951	0.6483	0.5922	0.6776	0.5722	**0.6379**	0.2128	0.3118	0.3040	0.3246	0.5759	0.6732	0.3351	0.3912
Decision Trees	0.9262	0.9305	0.9428	0.9515	0.9339	**0.9401**	0.8679	0.8802	0.8688	0.8817	0.9319	0.9537	0.8227	0.8536
Random Forest	0.9207	0.9305	0.9213	0.9480	0.9207	**0.9385**	0.8414	0.8770	0.8420	0.8783	0.9102	0.9537	0.7919	0.8536
Naive Bayes	0.7302	0.7607	0.7170	0.7741	0.7229	**0.7667**	0.4462	0.5337	0.4470	0.5346	0.7170	0.7605	0.4689	0.5206
K-Nearest Neighbor	0.6218	0.5981	0.5406	0.5994	0.5207	**0.5981**	0.1056	0.1969	0.1407	0.1974	0.5406	0.6565	0.3033	0.3636
Dataset 4														
Logistic Regression	0.6627	0.7081	0.5402	0.5982	0.4830	**0.5919**	0.1021	0.2340	0.1767	0.2859	0.5357	0.6429	0.3810	0.4184
Support Vector Machines	0.4311	0.5974	0.5348	0.6027	0.4536	**0.5360**	0.0731	0.1807	0.0803	0.2064	0.5357	0.6518	0.3810	0.4196
Decision Trees	0.9009	0.8938	0.8589	0.8884	0.8743	**0.8908**	0.7500	0.7817	0.7584	0.7821	0.8661	0.8839	0.7343	0.8230
Random Forest	0.9274	0.9258	0.8518	0.8839	0.8760	**0.9004**	0.7548	0.8015	0.7751	0.8086	0.8214	0.8839	0.7619	0.8230
Naive Bayes	0.6612	0.6967	0.6313	0.6429	0.6278	**0.6509**	0.2794	0.3168	0.2895	0.3352	0.6161	0.6696	0.4328	0.4497
K-Nearest Neighbor	0.5100	0.6297	0.5045	0.6161	0.4646	**0.6199**	0.0105	0.2430	0.0131	0.2454	0.5357	0.6161	0.3619	0.4107
Mean value of the measures to all the dataset														
Logistic Regression	0.7119	0.7172	0.6271	0.7119	0.6199	**0.6957**	0.2812	0.4055	0.3177	0.4226	0.6307	0.7221	0.4363	0.4809
Support Vector Machines	0.6577	0.6431	0.5906	0.6524	0.5246	**0.5971**	0.1738	0.2480	0.2334	0.2829	0.5677	0.6633	0.3800	0.4285
Decision Trees	0.8282	0.8296	0.8141	0.8380	0.8181	**0.8326**	0.6372	0.6657	0.6413	0.6672	0.8124	0.8365	0.6421	0.6748
Random Forest	0.8776	0.8598	0.7990	0.8759	0.8087	**0.8618**	0.6251	0.7255	0.6516	0.7328	0.7875	0.8739	0.6450	0.7072
Naive Bayes	0.7077	0.7246	0.6849	0.7132	0.6772	**0.6966**	0.3742	0.4125	0.3912	0.4355	0.6829	0.7233	0.4792	0.5058
K-Nearest Neighbor	0.6214	0.6254	0.5429	0.6524	0.5253	**0.6294**	0.1117	0.2645	0.1418	0.2738	0.5522	0.6676	0.3662	0.4277

Table 5
Comparison result between the imbalanced and synthetic balanced dataset using different learning techniques to show improvement with statistical analysis.

Measure	Bal+ LR vs Imb+ LR (Mean difference)	Bal+ SVM vs Imb+ SVM (Mean difference)	Bal+ DT vs Imb+ DT (Mean difference)	Bal+ RF vs Imb+ RF (Mean difference)	Bal+ NB vs Imb+ NB (Mean difference)	Bal+ KNN vs Imb+ KNN (Mean difference)
W/L/D (Precision)	2/2/0 (0.005)	2/2/0 (−0.014)	2/2/0 (0.001)	2/2/0 (−0.017)	3/1/0 (0.016)	2/2/0 (0.004)
W/L/D (Recall)	4/0/0 (0.084)	4/0/0 (0.061)	4/4/0 (0.023)	4/0/0 (0.076)	4/0/0 (0.024)	4/0/0 (0.109)
W/L/D (F1-Score)	4/0/0 (0.075)	4/0/0 (0.072)	4/0/0 (0.014)	4/0/0 (0.053)	4/0/0 (0.019)	4/0/0 (0.104)
W/L/D (MCC)	4/0/0 (0.104)	3/1/0 (0.049)	4/0/0 (0.025)	4/0/0 (0.081)	4/0/0 (0.044)	4/0/0 (0.132)
W/L/D (Cohen's Kappa)	4/0/0 (0.124)	4/0/0 (0.074)	4/0/0 (0.028)	4/0/0 (0.100)	4/0/0 (0.038)	4/0/0 (0.152)
W/L/D (G-Mean)	4/0/0 (0.091)	4/0/0 (0.096)	4/0/0 (0.024)	4/0/0 (0.086)	4/0/0 (0.040)	4/0/0 (0.115)
W/L/D (AUC)	4/0/0 (0.044)	4/0/0 (0.049)	4/0/0 (0.032)	4/0/0 (0.062)	4/0/0 (0.026)	4/0/0 (0.061)
Statistical Analysis						
p-value	0.0001	0.0082	0.0002	0.0006	0.0001	0.0002
Effect r	0.5237	0.3464	0.5196	0.4701	0.5526	0.5031
Significant Improvement	Yes	Yes	Yes	Yes	Yes	Yes

Table 6
MCC-based comparison to check the ability of the prediction model with the imbalanced and synthetic balanced dataset using all the six learning techniques.

Dataset	LR	SVM	DT	RF	NB	KNN
Dataset 1	0.1320	0.0651	0.0162	0.1064	0.0314	0.0989
Dataset 2	0.1257	−0.0140	0.0507	0.1485	0.0123	0.1400
Dataset 3	0.0524	0.0206	0.0129	0.0363	0.0876	0.0567
Dataset 4	0.1093	0.1261	0.0238	0.0335	0.0458	0.2323

compared to the imbalanced dataset. The negative value shows that the *CBReT* performs poorly for the specific learning technique corresponding to the conspicuous dataset. Table 6 shows that MCC improved for all the datasets with each machine-learning technique using the *CBReT* balanced dataset except the SVM classifier for Dataset 2. Here, we got the highest difference value of 0.148 for dataset 2 with the RF learning technique and the lowest value of 0.0123 for dataset 2 with NB learning techniques.

For Dataset 2 with SVM learning techniques we get the negative value, which shows that the prediction ability of the SVM learning technique is not improved for dataset 2. According to our experiments and analysis, dataset 2 performs poorly with the SVM. The reasoning behind this is that many instances of the minority class overlapped with the majority class. As we know, the *CBReT* generates synthetic points near the actual minority data points; therefore, the newly generated instances also overlap with the majority class's instances, as shown in Fig. 4. The performance of the SVM learning techniques depends on the separation of the classes. If the classes have overlapped, then SVM fails to draw a proper decision boundary between both classes,

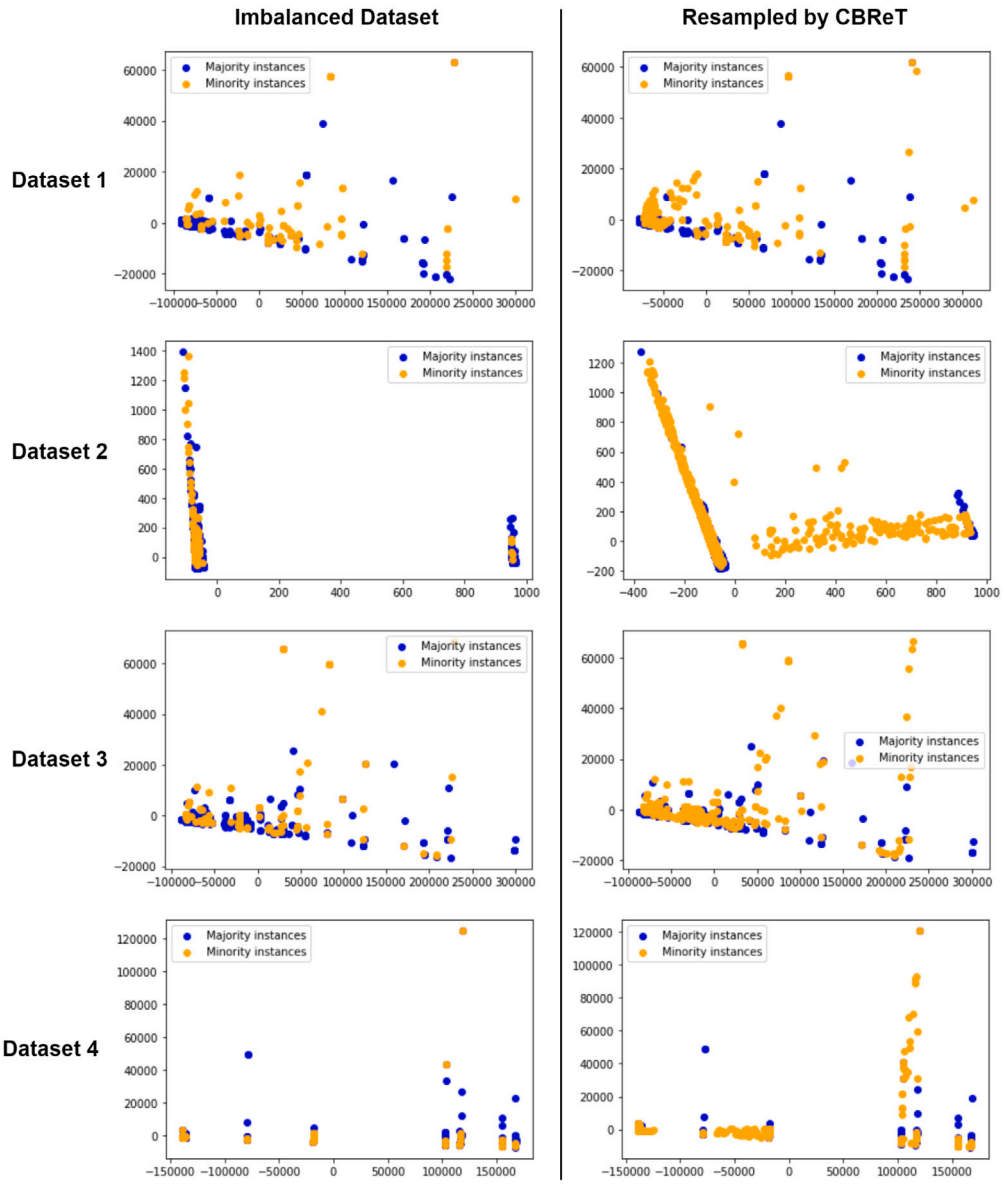

Fig. 4. Visualizing the original imbalanced data and synthetic balanced data using PCA.

and SVM finds difficulties in recognizing the class of the instance at the time of testing. Due to that, the performance of SVM in terms of MCC is decreased for dataset 2. Similarly, other included imbalance learning techniques also generate overlapped instances, which reduce the performance of the SVM, as shown in Fig. 5.

5.3. RQ3: How does CBReT technique improve the performance of the classifiers over the state-of-the-art approaches?

In recent studies, researchers like Khleel et al. [55], Fuyang Li et al. [56], Gupta et al. [57], and Rao et al. [8] have used oversampling approach such as (1) SMOTE (Synthetic Minority Over-sampling Technique), (2) SMOTEN (SMOTE- nominal features), (3) SVMSMOTE (SMOTE variant by using SVM classifier), (4) ADASYN (ADaptive SYNthetic sampling method), and (5) ROS (Random Oversampling) to overcome the imbalance issue in the code smell detection task. We have compared the *CBReT* technique with these oversampling approaches to find the answer to this research question.

We build code smell prediction models to compare the performance of the *CBReT* technique with the existing techniques. In this model, we have used each technique one by one to perform balancing on the same training data. After the training, the prediction model have been tested using the test dataset, and results are reported in various selective measures. The above experiment has been repeated ten times for the single dataset, and the mean of the measures is stored at the end. The whole process has been repeated for all the included resampling approaches. After completing the experiments, the summarized results have been stored in Tables 7 and 8 for observation. The improved results have been written in dark letters or numbers. The following are the observations drawn from the experiments:

1. The *CBReT* technique for the LR learning technique performs better for the f1-score, with the highest value of 0.74 for Dataset 3 and the lowest value of 0.59 for Dataset 4 as compared to other state-of-the-art. It also improved the MCC and Cohen's kappa parameters with the highest value of 0.50 and 0.50 for Dataset 3 and the lowest value of 0.28 and 0.23 for Dataset 4, respectively. Regarding precision and recall, LR also shows improved results with the *CBReT* techniques as compared to the others for all four datasets except Dataset 4, where recall lagged by ADASYN by

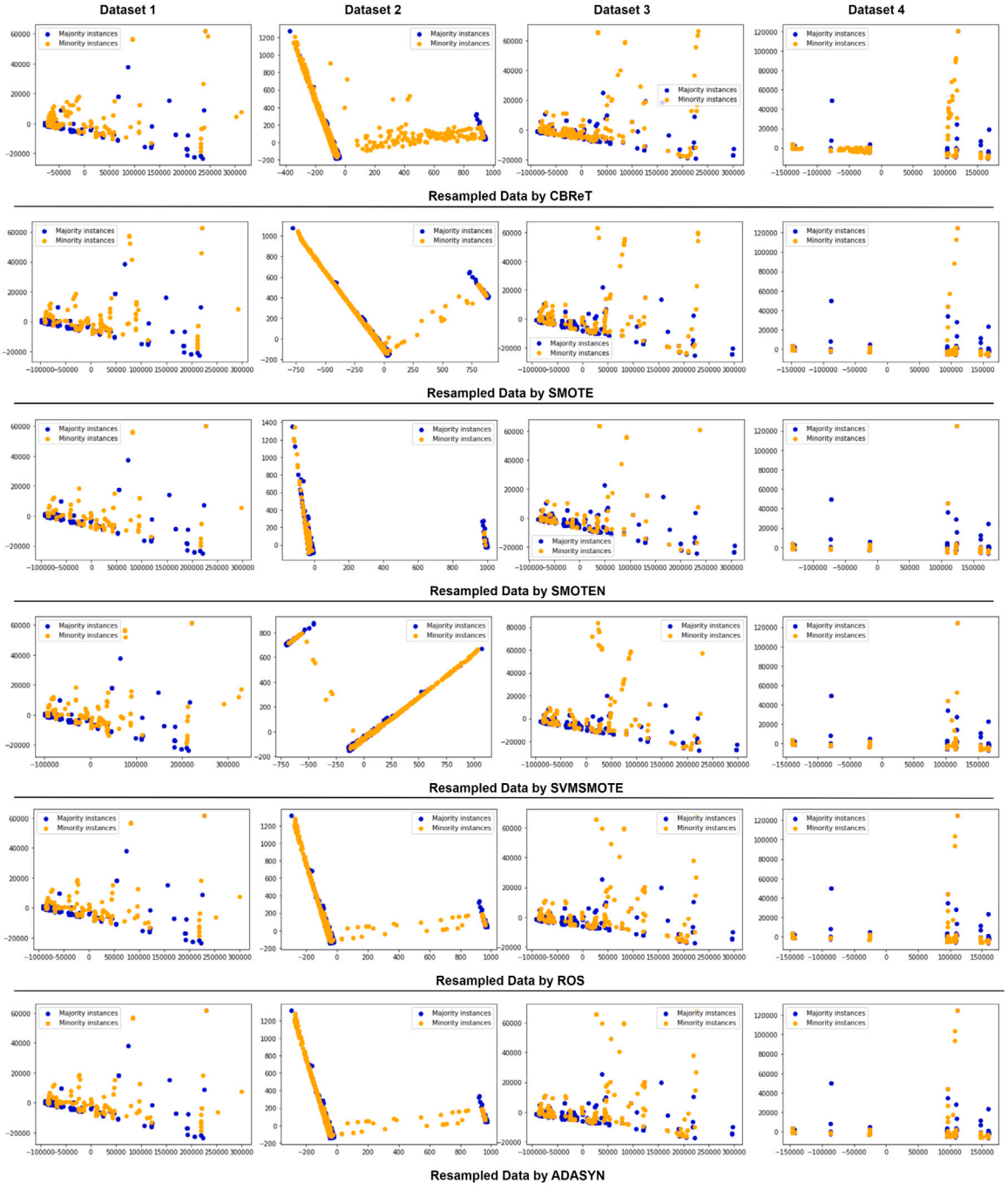

Fig. 5. Visualizing the original imbalanced data and resampled data (by the *CBReT*, SMOTE, SMOTEN, SVMSMOTE, ROS, and ADASYN) using PCA.

the value 0.0170. However, at the same time, the precision leads with 0.950, and the overall f1-score improves by 0.572.

2. We have also found that KNN performs better with *CBReT* than state-of-the-art. In the Table 7, we can see that *CBReT* gets improved results in all areas. Regarding the f1-score, SVM gives enhanced results with the highest value of 0.71 for Dataset 2 and the lowest value of 0.58 for Dataset 1. Similarly, MCC improved with the highest value of 0.46 for Dataset 2 and the lowest value of 0.18 for Dataset 1. Further, Cohen's kappa value has also improved, with the highest value of 0.43 for Dataset 2 and the lowest value of 0.18 for Dataset 1. Also, regarding precision and recall, KNN shows improvement for all the datasets.
   In summary, the KNN shows improvement with the CBReT technique for all datasets as compared to the state-of-the-art.
3. While comparing the results of the SVM learning technique, we again found that *CBReT* gives better results in terms of the f1-score with the highest and lowest values of 0.65 for Dataset 2 and 0.53 for Dataset 4, respectively. However, *CBReT* SVM improves MCC for all datasets except Dataset 3, where ROS beats it by 0.0037. At the same time, for Cohen's kappa value, *CBReT* beats the ROS by 0.0273. Also, for the *CBReT*, Cohen's kappa shows better improvement for all datasets, with the highest and lowest value of 0.32 for Dataset 2 and 0.18 for Dataset 4, respectively. Analyzing the precision, we get improved results by the *CBReT* SVM for all datasets except Datasets 1 and 2, where ADASYN SVM gets leads, but at the same time, *CBReT* SVM's recall value is increased.

Table 7
Comparison Results between Proposed method and other included imbalanced learning approaches using all six learning techniques.

Dataset	Performance measure	*CBReT* (LR)	SMOTE (LR)	SMOTEN (LR)	SVM-S (LR)	ADASYN (LR)	ROS (LR)	*CBReT* (KNN)	SMOTE (KNN)	SMOTEN (KNN)	SVM-S (KNN)	ADASYN (KNN)	ROS (KNN)
	precision	**0.7388**	0.7276	0.7126	0.7276	0.7339	0.7150	**0.5927**	0.5749	0.5616	0.5643	0.5711	0.5617
	recall	**0.7355**	0.7284	0.7146	0.7284	0.7331	0.7157	**0.5932**	0.5734	0.5581	0.5620	0.5710	0.5615
Dataset 1	f1-score	**0.7364**	0.7261	0.7113	0.7261	0.7322	0.7146	**0.5865**	0.5607	0.5417	0.5452	0.5635	0.5520
	Cohen_Kappa	**0.4733**	0.4537	0.4245	0.4537	0.4653	0.4298	**0.1819**	0.1422	0.1114	0.1194	0.1385	0.1195
	MCC	**0.4743**	0.4560	0.4271	0.4560	0.4669	0.4307	**0.1859**	0.1483	0.1195	0.1263	0.1420	0.1232
	precision	**0.6830**	0.6224	0.6139	0.6224	0.6216	0.6306	**0.6809**	0.6429	0.6566	0.6426	0.6385	0.6317
	recall	**0.7429**	0.7106	0.6122	0.7106	0.7157	0.7309	**0.8010**	0.7792	0.6451	0.7799	0.7787	0.7598
Dataset 2	f1-score	**0.7052**	0.6417	0.6123	0.6417	0.6405	0.6514	**0.7130**	0.6642	0.6499	0.6637	0.6572	0.6496
	Cohen_Kappa	**0.4128**	0.2971	0.2251	0.2971	0.2968	0.3185	**0.4360**	0.3518	0.3002	0.3512	0.3409	0.3240
	MCC	**0.4212**	0.3209	0.2257	0.3209	0.3238	0.3471	**0.4663**	0.3994	0.3011	0.3995	0.3928	0.3698
	precision	**0.7387**	0.7107	0.7275	0.7107	0.7156	0.7057	**0.5981**	0.5810	0.5658	0.5969	0.5762	0.5896
	recall	**0.7712**	0.7350	0.7552	0.7350	0.7505	0.7393	**0.5994**	0.5691	0.5423	0.5807	0.5938	0.5899
Dataset 3	f1-score	**0.7493**	0.7179	0.7365	0.7179	0.7238	0.7139	**0.5981**	0.5716	0.5397	0.5846	0.5645	0.5892
	Cohen_Kappa	**0.5016**	0.4392	0.4759	0.4392	0.4536	0.4336	**0.1969**	0.1473	0.0983	0.1738	0.1577	0.1791
	MCC	**0.5089**	0.4449	0.4818	0.4449	0.4647	0.4437	**0.1974**	0.1495	0.1055	0.1768	0.1690	0.1794
	precision	**0.7081**	0.5933	0.5452	0.5933	0.6131	0.5952	**0.6297**	0.5907	0.5699	0.5745	0.5881	0.5943
	recall	0.5982	0.5911	0.5464	0.5911	**0.6089**	0.5991	**0.6161**	0.5884	0.5616	0.5720	0.5902	0.5955
Dataset 4	f1-score	**0.5919**	0.5290	0.5358	0.5290	0.5347	0.5469	**0.6199**	0.5881	0.5596	0.5720	0.5886	0.5939
	Cohen_Kappa	**0.2340**	0.1488	0.0921	0.1488	0.1770	0.1639	**0.2430**	0.1782	0.1286	0.1458	0.1778	0.1893
	MCC	**0.2859**	0.1840	0.0916	0.1840	0.2212	0.1941	**0.2454**	0.1790	0.1310	0.1464	0.1782	0.1898
		CBReT (SVM)	SMOTE (SVM)	SMOTEN (SVM)	SVM-S (SVM)	ADASYN (SVM)	ROS (SVM)	*CBReT* (DT)	SMOTE (DT)	SMOTEN (DT)	SVM-S (DT)	ADASYN (DT)	ROS (DT)
	precision	**0.6889**	0.6627	0.6625	0.6627	0.6884	0.7010	**0.8579**	0.8287	0.8476	0.8287	0.8138	0.8296
	recall	**0.6142**	0.5951	0.5748	0.5951	0.6020	0.5780	**0.8519**	0.8298	0.8495	0.8298	0.8156	0.8293
Dataset 1	f1-score	**0.5578**	0.5466	0.5018	0.5466	0.5522	0.5066	**0.8541**	0.8270	0.8455	0.8270	0.8128	0.8269
	Cohen_Kappa	**0.2202**	0.1878	0.1478	0.1878	0.2079	0.1579	**0.7084**	0.6551	0.6921	0.6551	0.6265	0.6550
	MCC	**0.2854**	0.2406	0.2010	0.2406	0.2707	0.2401	**0.7097**	0.6586	0.6971	0.6586	0.6294	0.6589
	precision	0.6342	0.6129	0.5990	0.6129	**0.6523**	0.6412	**0.6362**	0.6157	0.6105	0.6157	0.5997	0.6143
	recall	**0.7426**	0.7091	0.6418	0.7091	0.6997	0.7063	**0.6604**	0.6238	0.5960	0.6238	0.6504	0.6349
Dataset 2	f1-score	**0.6569**	0.6176	0.5960	0.6176	0.6482	0.6428	**0.6453**	0.6186	0.6008	0.6186	0.6110	0.6227
	Cohen_Kappa	**0.3282**	0.2658	0.2097	0.2658	0.3117	0.3095	**0.2923**	0.2380	0.2028	0.2380	0.2323	0.2464
	MCC	**0.3607**	0.3051	0.2298	0.3051	0.3382	0.3357	**0.2951**	0.2389	0.2050	0.2389	0.2441	0.2482
	precision	0.6483	0.6943	0.6836	0.6943	**0.7454**	0.6894	**0.9305**	0.9241	0.9081	0.9241	0.9166	0.9187
	recall	**0.6776**	0.6444	0.6144	0.6444	0.6210	0.6570	**0.9515**	0.9385	0.9058	0.9385	0.9232	0.9276
Dataset 3	f1-score	**0.6379**	0.6089	0.5858	0.6089	0.5606	0.6053	**0.9401**	0.9301	0.9069	0.9301	0.9196	0.9228
	Cohen_Kappa	**0.3118**	0.2696	0.2341	0.2696	0.2417	0.2845	**0.8802**	0.8604	0.8138	0.8604	0.8392	0.8456
	MCC	0.3246	0.3186	0.2755	0.3186	0.3135	**0.3283**	**0.8817**	0.8622	0.8139	0.8622	0.8397	0.8462
	precision	**0.5974**	0.5631	0.5528	0.5631	0.5438	0.5358	**0.8938**	0.8805	0.8809	0.8805	0.8719	0.8759
	recall	**0.6027**	0.5705	0.5750	0.5705	0.5732	0.5714	**0.8884**	0.8536	0.8366	0.8536	0.8580	0.8589
Dataset 4	f1-score	**0.5360**	0.5324	0.5105	0.5324	0.5210	0.4585	**0.8908**	0.8637	0.8514	0.8637	0.8637	0.8647
	Cohen_Kappa	**0.1807**	0.1372	0.1359	0.1372	0.1330	0.1275	**0.7817**	0.7283	0.7048	0.7283	0.7277	0.7301
	MCC	**0.2064**	0.1506	0.1531	0.1506	0.1499	0.1661	**0.7821**	0.7332	0.7155	0.7332	0.7297	0.7342
		CBReT (RF)	SMOTE (RF)	SMOTEN (RF)	SVM-S (RF)	ADASYN (RF)	ROS (RF)	*CBReT* (NB)	SMOTE (NB)	SMOTEN (NB)	SVM-S (NB)	ADASYN (NB)	ROS (NB)
	precision	**0.9150**	0.8614	0.8724	0.8614	0.8757	0.8724	**0.7255**	0.6903	0.6642	0.6903	0.6903	0.6994
	recall	**0.9110**	0.8610	0.8757	0.8610	0.8710	0.8692	**0.6771**	0.6592	0.6416	0.6592	0.6592	0.6646
Dataset 1	f1-score	**0.9127**	0.8611	0.8728	0.8611	0.8726	0.8703	**0.6349**	0.6238	0.6097	0.6238	0.6238	0.6281
	Cohen_Kappa	**0.8254**	0.7222	0.7460	0.7222	0.7455	0.7408	**0.3306**	0.2990	0.2670	0.2990	0.2990	0.3088
	MCC	**0.8260**	0.7223	0.7481	0.7223	0.7468	0.7415	**0.3997**	0.3481	0.3050	0.3481	0.3481	0.3622
	precision	0.6680	0.6626	**0.7878**	0.6626	0.6714	0.6745	**0.7155**	0.7126	0.4940	0.7126	0.7047	0.6971
	recall	**0.7609**	0.6512	0.5354	0.6512	0.6631	0.6582	0.7589	0.7519	0.4880	0.7519	**0.7671**	0.7663
Dataset 2	f1-score	**0.6955**	0.6558	0.5398	0.6558	0.6662	0.6648	**0.7340**	0.7287	0.3765	0.7287	0.7281	0.7225
	Cohen_Kappa	**0.3982**	0.3120	0.1153	0.3120	0.3328	0.3301	**0.4688**	0.4584	−0.005	0.4584	0.4585	0.4479
	MCC	**0.4183**	0.3131	0.1986	0.3131	0.3338	0.3316	**0.4723**	0.4623	−0.017	0.4623	0.4671	0.4579
	precision	**0.9305**	0.9303	0.9201	0.9303	0.9290	0.9242	**0.7607**	0.7378	0.7545	0.7378	0.7484	0.7515
	recall	**0.9480**	0.9444	0.9382	0.9444	0.9423	0.9433	**0.7741**	0.7279	0.7518	0.7279	0.7431	0.7475
Dataset 3	f1-score	**0.9385**	0.9367	0.9282	0.9367	0.9352	0.9327	**0.7667**	0.7323	0.7530	0.7323	0.7455	0.7493
	Cohen_Kappa	**0.8770**	0.8734	0.8566	0.8734	0.8703	0.8654	**0.5337**	0.4649	0.5062	0.4649	0.4912	0.4987
	MCC	**0.8783**	0.8745	0.8580	0.8745	0.8711	0.8672	**0.5346**	0.4655	0.5063	0.4655	0.4915	0.4989
	precision	**0.9258**	0.8915	0.8938	0.8915	0.8790	0.8965	**0.6967**	0.6013	0.5579	0.6013	0.6234	0.6287
	recall	**0.8839**	0.8321	0.8330	0.8321	0.8330	0.8598	**0.6429**	0.5946	0.5429	0.5946	0.6214	0.6125
Dataset 4	f1-score	**0.9004**	0.8514	0.8528	0.8514	0.8479	0.8740	**0.6509**	0.5350	0.4307	0.5350	0.5601	0.5275
	Cohen_Kappa	**0.8015**	0.7057	0.7084	0.7057	0.6981	0.7489	**0.3170**	0.1641	0.0646	0.1641	0.2120	0.1869
	MCC	**0.8086**	0.7207	0.7239	0.7207	0.7098	0.7553	**0.3352**	0.1948	0.0992	0.1948	0.2440	0.2383

Abbreviations: SVM-S: SVMSMOTE.

4. The *CBReT* also improved the performance of the DT as compared to the other existing methods. In terms of the f1-score, *CBReT* DT gives the highest value of 0.94 for Dataset 3 and the lowest value of 0.64 for Dataset 2. Similarly, *CBReT* DT improves the results for MCC with the highest and lowest values of 0.88 for Dataset 3 and 0.29 for Dataset 2, respectively. At the same time, Cohen's kappa value has also improved with the highest and lowest values of 0.88 for Dataset 3 and 0.29 for Dataset 2, respectively. *CBReT* DT also improves the precision and recall of all the datasets.
5. During the assessment of the RF, we see that *CBReT* improves the f1-score for all the datasets, with the highest value of 0.93 for

Table 8
Comparison Results based on the G-mean and AUC between Proposed method and other imbalanced learning approaches using all six learning techniques.

Dataset	Performance Measure	*CBReT* (LR)	SMOTE (LR)	SMOTEN (LR)	SVM-S (LR)	ADASYN (LR)	ROS (LR)	*CBReT* (KNN)	SMOTE (KNN)	SMOTEN (KNN)	SVM-S (KNN)	ADASYN (KNN)	ROS (KNN)
Dataset 1	G-Mean	**0.7421**	0.7230	0.7316	0.7230	0.7271	0.7171	**0.6009**	0.5732	0.5489	0.5770	0.5655	0.5655
	AUC	**0.7216**	0.7107	0.7179	0.7107	0.7142	0.7042	**0.6188**	0.6020	0.5878	0.6051	0.5964	0.5964
Dataset 2	G-Mean	**0.7324**	0.7141	0.6134	0.7244	0.7146	0.7146	**0.7971**	0.7815	0.6371	0.7471	0.7676	0.7851
	AUC	**0.2788**	0.2390	0.1761	0.2391	0.2235	0.2235	**0.3175**	0.2616	0.2031	0.2526	0.2545	0.2673
Dataset 3	G-Mean	**0.7712**	0.7224	0.7359	0.7224	0.7413	0.7466	**0.6565**	0.5837	0.5349	0.5756	0.5891	0.5862
	AUC	**0.5047**	0.4622	0.4715	0.4622	0.4671	0.4641	**0.3636**	0.3250	0.2919	0.3177	0.3267	0.3220
Dataset 4	G-Mean	**0.6429**	0.5893	0.5714	0.6071	0.589286	0.5893	**0.6161**	0.5893	0.5536	0.5804	0.5625	0.5625
	AUC	**0.4184**	0.3805	0.3706	0.3905	0.380102	0.3815	**0.4107**	0.3867	0.3639	0.3798	0.3686	0.3686
Dataset	Performance Measure	*CBReT* (SVM)	SMOTE (SVM)	SMOTEN (SVM)	SVM-S (SVM)	ADASYN (SVM)	ROS (SVM)	*CBReT* (DT)	SMOTE (DT)	SMOTEN (DT)	SVM-S (DT)	ADASYN (DT)	ROS (DT)
Dataset 1	G-Mean	**0.6210**	0.5905	0.5910	0.5905	0.6070	0.5853	**0.8551**	0.8161	0.8340	0.8161	0.8106	0.8055
	AUC	**0.6406**	0.6175	0.6189	0.6175	0.6262	0.6124	**0.8324**	0.7973	0.8189	0.7973	0.7946	0.7884
Dataset 2	G-Mean	0.7073	**0.7139**	0.6271	0.6834	0.6980	0.6739	**0.6534**	0.6212	0.5907	0.6212	0.6178	0.6288
	AUC	**0.2627**	0.2112	0.1930	0.2470	0.2195	0.2120	**0.1903**	0.1731	0.1626	0.1731	0.1635	0.1776
Dataset 3	G-Mean	**0.6732**	0.6030	0.5759	0.6030	0.5841	0.5841	**0.9537**	0.9102	0.9102	0.9319	0.9319	0.9319
	AUC	**0.3912**	0.3544	0.3351	0.3544	0.3502	0.3502	**0.8536**	0.7919	0.7919	0.8227	0.8227	0.8227
Dataset 4	G-Mean	**0.6518**	0.5446	0.5089	0.5446	0.6071	0.5804	**0.8839**	0.8661	0.8304	0.8661	0.8571	0.8572
	AUC	**0.4196**	0.3608	0.3374	0.3608	0.3905	0.3741	**0.8230**	0.7343	0.7164	0.7343	0.7363	0.7364
Dataset	Performance Measure	*CBReT* (RF)	SMOTE (RF)	SMOTEN (RF)	SVM-S (RF)	ADASYN (RF)	ROS (RF)	*CBReT* (NB)	SMOTE (NB)	SMOTEN (NB)	SVM-S (NB)	ADASYN (NB)	ROS (NB)
Dataset 1	G-Mean	**0.9083**	0.8634	0.8874	0.8634	0.8631	0.8696	**0.6857**	0.6601	0.6449	0.6601	0.6598	0.6598
	AUC	**0.8777**	0.8411	0.8710	0.8411	0.8404	0.8478	**0.7083**	0.6785	0.6619	0.6785	0.6778	0.6778
Dataset 2	G-Mean	**0.7499**	0.6456	0.5363	0.6456	0.6532	0.6544	**0.7773**	0.7310	0.4993	0.7498	0.7561	0.7561
	AUC	**0.2744**	0.2280	0.1457	0.2280	0.2316	0.2357	**0.3448**	0.2971	0.1086	0.3169	0.3162	0.3162
Dataset 3	G-Mean	**0.9537**	0.9184	0.9319	0.9184	0.9455	0.9319	**0.7741**	0.7170	0.7605	0.7388	0.7387	0.7388
	AUC	**0.8536**	0.8262	0.8227	0.8262	0.8213	0.8227	0.5116	0.4689	**0.5206**	0.4945	0.4945	0.4945
Dataset 4	G-Mean	**0.8839**	0.8304	0.8571	0.8304	0.8393	0.8661	**0.6696**	0.5625	0.5714	0.5625	0.6339	0.5446
	AUC	**0.8230**	0.7164	0.7363	0.7164	0.7133	0.7629	**0.4497**	0.3655	0.3687	0.3655	0.4127	0.3553

Abbreviations: SVM-S: SVMSMOTE.

Dataset 3 and the lowest value of 0.69 for Dataset 2. Regarding the MCC, *CBReT* gives better results for all the datasets, with the highest value of 0.87 for Dataset 3 and the lowest value of 0.41 for Dataset 2. Whereas *CBReT* improves Cohen's kappa for the RF learning technique with the highest value of 0.87 for Dataset 3 and the lowest value of 0.39 for Dataset 2. Also, *CBReT* gives better results compared to the other techniques for all four datasets except Dataset 2, where SMOTEN precision gets leads. However, at the same time, *CBReT* recall performs better than the SMOTEN and other state-of-the-art.

6. Concerning the NB, the *CBReT* shows better performance for the f1-score, with the highest value of 0.76 for Dataset 3 and the lowest value of 0.63 for Dataset 1. *CBReT* also improves the performance of the MCC regarding all datasets, with the highest value of 0.53 for Dataset 3 and the lowest value of 0.33 for Dataset 4. We have also observed the improvement in Cohen's kappa with the highest and lowest value of 0.53 (for Dataset 3) and 0.31 (for Dataset 4), respectively. We also get improvements in recall for all the datasets except Dataset 2, where ADASYN NB leads with 0.0082. However, at the same time, we see an improvement in precision for all the datasets compared to the other imbalanced learning techniques.
7. Furthermore, we also evaluated the code smell prediction models based on the G-mean and AUC performance measures, and the results are shown in Table 8. It is evident that the performance of the code smell prediction models has improved with the CBReT technique compared to the state-of-the-art for all the datasets. The highest G-mean of 0.9537 is achieved for Dataset 3, while the lowest G-mean, 0.6009, is observed for Dataset 1. In terms of AUC, Dataset 1 attained the highest value of 0.8777, while the lowest AUC of 0.1903 was recorded for Dataset 2.
8. For the additional evidences, we also perform a statistical analysis test between the proposed *CBReT* and other included imbalanced learning techniques; results are shown in Table 9. Table 9 shows that *CBReT* improves all learning techniques' performance compared to other state-of-the-art. We can see a significant improvement in the results of all six learning techniques using the proposed *CBReT* technique compared to others.

Therefore, we can conclude that the *CBReT* improves the performance of all basic machine learning techniques compared to the state-of-the-art for the considered code smell datasets.

5.4. Comparison with existing state-of-the-art works

We compare the performance of the *CBReT* technique with the other state-of-the-art approaches. Table 10 summarizes the comparative analysis.

Table 10 provides a theoretical comparison between CBReT and other state-of-the-art approaches. The results have been directly extracted from their respective articles. As a result, variations in the dataset, experimental environment, and setups may exist.

In [12], the authors utilized a dataset comprising five types of code smells. To address the issue of data imbalance, they incorporated five data balancing techniques and compared the performance of prediction models against baseline models with no balancing. While acknowledging certain limitations, they observed that the oversampling method SMOTE marginally improved prediction performance. Consequently, they concluded that further work is needed to enhance classifier performance in code smell detection. In [58], the authors employed a Convolutional Neural Network (CNN) for binary classification across eight types of code smell datasets. Their findings revealed that the CNN model achieved the highest f1-score of 0.810 for detecting Contrived Complexity code smell. Similarly, F Li et al. [59] conducted multiple experiments to assess the performance of SMOTE and other oversampling techniques on four code smell datasets. They discovered that several oversampling methods outperformed SMOTE. We have selected the top-performing oversampling approach from their findings to compare with our proposed approach. In [41], the authors have employed bagging and XGBoost classifiers with the SMOTEENN (a combination of

Table 9
Statistical analysis results for comparison between proposed *CBReT* and other included imbalance learning techniques.

Comparision group	Measures	LR	SVM	DT	RF	NB	KNN
CBReT and SMOTE	*p*-value	0.00001	0.00234	0.00001	0.00002	0.00001	0.00001
	r-Value	0.61859	0.40827	0.61859	0.58560	0.61859	0.61447
	Significant Difference	Yes	Yes	Yes	Yes	Yes	Yes
CBReT and SMOTEN	*p*-value	0.00001	0.00005	0.00001	0.00041	0.00001	0.00001
	r-Value	0.61859	0.56498	0.61034	0.48250	0.61859	0.60622
	Significant Difference	Yes	Yes	Yes	Yes	Yes	Yes
CBReT and SVMSMOTE	*p*-value	0.00001	0.00234	0.00001	0.00002	0.00001	0.00001
	r-Value	0.61859	0.40827	0.61859	0.58560	0.61859	0.60622
	Significant Difference	Yes	Yes	Yes	Yes	Yes	Yes
CBReT and ROS	*p*-value	0.00002	0.01033	0.00001	0.00004	0.00001	0.00001
	r-Value	0.58972	0.33404	0.61859	0.56910	0.60622	0.61859
	Significant Difference	Yes	Yes	Yes	Yes	Yes	Yes
CBReT and ADASYN	*p*-value	0.00001	0.00022	0.00001	0.00006	0.00001	0.00001
	r-Value	0.61447	0.50724	0.61859	0.55261	0.60622	0.61859
	Significant Difference	Yes	Yes	Yes	Yes	Yes	Yes

SMOTEN and Edited Nearest Neighbor) imbalance learning techniques to detect two types of code smells. They have also utilized CuBERT embedding techniques to extract features directly from Java source codes. Likewise, in [60], six conventional machine learning techniques employed to detect two types of code smells in Python. In the table, we have included only the best-performing classifier for each dataset. In [61], authors applied deep direct learning with transfer learning to detect four types of code smells. They employed CNN, RNN, and Auto Encoder techniques, which taking raw source code as an input and predict code smells. The authors concluded that the Auto Encoder outperformed the CNN and RNN in this context.

Based on Table 10, it is clear that the proposed work, *CBReT*, demonstrates superior performance across all performance metrics. However, for the God Class dataset, RF with ENN (presented by F. Li et al. [59]) outperforms *CBReT*. This can be attributed to the dataset used by F. Li et al. which has a ratio of 140 : 559 (smelly instances:non-smelly instances), whereas our dataset has a ratio of 251 : 2049 (smelly instances: non-smelly instances). There is a significant difference between the ratios (smelly and non-smelly instances) in these two datasets perhaps this is the reason for the slight performance decrease in the case of the God class dataset. The CNN based

Moreover, we can observe that the deep learning models, such as CNN, used for comparison with CBReT do not perform better. This is because deep learning models require a large number of instances with high-dimensional features to learn the dataset's distribution. However, the existing code smell benchmark datasets' size is limited.

5.5. Discussion

Previous literature indicates that many researchers have employed SMOTE and its variations to address the imbalance issue in code smell detection models. SMOTE relies on a distance-based random point generation technique, selecting two nearest points and generating random instances between them. However, SMOTE does not consider the data's distribution during instance generation.

On the other hand, GANs (Generative Adversarial Networks) and deep learning models learn the data distribution and generate synthetic instances accordingly. However, deep learning models require more features and instances to grasp the data distribution effectively. Unfortunately, the currently available code smell datasets offer limited features and instances, making deep learning less effective in this context.

In contrast, *CBReT* utilizes a distance-based algorithm that accounts for the data's distribution when generating synthetic instances. Additionally, the PSI phase (Section 3.3) validates these generated instances. If a synthetic instance is deemed valid, it will be accepted by PSI; otherwise, it will be discarded, as explained in Section 3.5. The PSI phase controls the area of the synthetic instances, preventing them from disturbing the decision or separation boundary between the smelly and non-smelly classes. This advantage makes the *CBReT* technique more reliable and robust in addressing the data imbalance issue in overlapped code smell datasets.

CBReT Performance in Overlapped Datasets: To address the issue of instance overlap, we employed two types of datasets: (1) Highly overlapped and (2) Moderately overlapped. The highly overlapped datasets include Dataset 2 and Dataset 4, while the moderately overlapped datasets consist of Dataset 1 and Dataset 3. The distribution of instances is visually depicted in Fig. 4. From Fig. 4, it is noticeable that in Datasets 2 and 4, a significant number of instances from both minority and majority classes overlap with each other. Conversely, in Datasets 1 and 3, there is less overlap between some minority and majority instances.

In our experimental results, we observed that the performance of classifiers, measured in terms of the F1-score, is lower for the highly overlapped imbalanced dataset than the moderately overlapped imbalanced datasets, shown in Table 4. However, upon synthesizing minority instances to balance the datasets using the *CBReT* and state-of-the-art techniques, we noted a significant improvement in classifier performance with *CBReT*, even in the presence of highly overlapped datasets. *CBReT* significantly enhances classifier performance in the range of 0.5360 to 0.9004 for highly overlapped datasets, demonstrating superior results compared to the state-of-the-art techniques, which show improvements ranging from 0.0036 to 0.0908. Similarly, for moderately overlapped datasets, *CBReT* improves classifier performance in the range of 0.5578 to 0.9401, surpassing state-of-the-art techniques by 0.0018 to 0.0399, as shown in Table 11. The table presents the differences of the f1-score between the *CBReT* technique and the best of the state-of-the-art with respect to each dataset. In the table, the positive sign indicates that *CBReT* improved the F1-score compared to the best of the state-of-the-art.

In summary, based on the above analysis, *CBReT* consistently outperforms state-of-the-art techniques, even in scenarios where the dataset exhibits high overlap between instances.

6. Conclusions and future work

In this paper, we have introduced an innovative imbalance learning technique, *CBReT*, designed to enhance the performance of code smell prediction models. The *CBReT* generates synthetic data for the minority class, utilizing the inherent dataset distribution. During the synthesis process, CBReT verifies the validity of each newly generated instance, ensuring that only valid instances are included. This quality control enhances the overall dataset quality and does not compromise the class boundaries, even when dealing with overlapping data. The code smell

Table 10
Comparison of presented work with other related work Code Smell detection.

Related work	Used dataset	Learning model	Precision	Recall	F1-score	MCC	Cohen's Kappa
F Pecorelli et al. [12]	God Class	NB with SMOTE	0.26	0.93	0.41	0.49	–
	Spaghetti Code	NB with SMOTE	0.16	0.34	0.22	0.22	–
	Class Data Should Be Private	NB with ClassBalancer	0.23	0.55	0.33	0.35	–
	Complex Class	NB with SMOTE	0.26	0.65	0.37	0.4	–
	Long Method	NB with No-balancing	0.15	0.56	0.23	0.28	–
T Lin et al. [58]	Long Method	CNN	0.52	0.67	0.75	–	0.63
	Lazy Class	CNN	0.62	0.67	0.61	–	0.63
	Speculative Generality	CNN	0.71	0.73	0.68	–	0.64
	Refused Bequest	CNN	0.69	0.70	0.71	–	0.67
	Duplicated code	CNN	0.54	0.56	0.59	–	0.58
	Contrived complexity	CNN	0.78	0.79	0.81	–	0.80
	Shotgun surgery	CNN	0.59	0.59	0.501	–	0.60
	Uncontrolled side effects	CNN	0.80	0.79	0.80	–	0.81
F Li et al. [59]	Data Class	DT with CNN	0.87	0.77	0.80	0.54	–
	God Class	RF with ENN	0.86	0.80	0.81	0.51	–
	Feature Envy	RF with BSMOTE	0.89	0.86	0.87	0.70	–
	Long Method	LR with ROS	0.88	0.85	0.86	0.67	–
A Kovačević et al. [41]	God Class	Bagging (SVM classifier) with SMOTEENN	0.48	0.58	0.53	–	–
	Long Method	XGBoost with SMOTEENN	0.70	0.81	0.75	–	–
R Sandouka et al. [60]	Large Class (Python)	RF	–	–	–	0.77	–
	long Method (Python)	DT	–	–	–	0.90	–
T Sharma et al. [61]	Complex Method	Auto Encoder	0.60	0.68	0.64	0.67	–
	Complex Conditional	Auto Encoder	0.20	0.20	0.20	0.21	–
	Feature Envy	Auto Encoder	0.18	0.24	0.21	0.22	–
	Multifaceted abstraction	Auto Encoder	0.03	0.14	0.05	0.06	–
S Boutaib et al. [35]	Blob	ADIODE	–	–	94.5	–	–
	Data Class	ADIODE	–	–	90.07	–	–
	Feature Envy	ADIODE	–	–	89.53	–	–
	Long Method	ADIODE	–	–	88.35	–	–
	Long Parameter	ADIODE	–	–	88.25	–	–
CBReT	**Data Class**	**RF**	**0.91**	**0.91**	**0.91**	**0.82**	**0.82**
	God Class	**NB**	**0.71**	**0.75**	**0.73**	**0.47**	**0.46**
	Feature Envy	**DT**	**0.93**	**0.95**	**0.94**	**0.88**	**0.88**
	Long Parameter	**RF**	**0.92**	**0.88**	**0.90**	**0.80**	**0.80**

Table 11
Difference of the F1-Score between the *CBReT* technique and best of the state-of-the-art with respect to each dataset.

Dataset	LR	SVM	DT	RF	NB	KNN
Dataset 1	+0.0042	+0.0056	+0.0086	+0.0399	+0.0068	+0.0230
Dataset 2	+0.0537	+0.0087	+0.0227	+0.0293	+0.0052	+0.0488
Dataset 3	+0.0128	+0.0289	+0.0099	+0.0018	+0.0136	+0.0089
Dataset 4	+0.0450	+0.0036	+0.0261	+0.0264	+0.0908	+0.0260

detection model has been evaluated on the four benchmark code smell datasets. Subsequently, we have conducted a comprehensive comparative analysis, testing the *CBReT* technique against other state-of-the-art works. The results demonstrate that the *CBReT* technique significantly improves the detection model's performance, enhancing at least 0.18% and up to 9.08% as compared to other imbalance learning techniques for code smell datasets. Furthermore, our technique exhibited robust performance compared to other state-of-the-art works. In the future, we plan to explore the credibility of the GAN-based models for generating synthetic data for code smell datasets.

CRediT authorship contribution statement

Praveen Singh Thakur: Writing – original draft, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Mahipal Jadeja:** Writing – review & editing, Validation, Supervision, Investigation, Formal analysis, Conceptualization. **Satyendra Singh Chouhan:** Writing – review & editing, Validation, Supervision, Investigation, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We used a publicly available dataset. Here's the link: https://github.com/hjamaan/IST2021-CodeSmellStackingEnsemble/tree/main/Datasets/Originaland https://codeocean.com/capsule/5256791/tree/v1l.

Appendix

See Tables A.12–A.14.

Table A.12
Comparison Results between imbalanced dataset and synthetic balanced dataset using the different learning classifiers by employing 10-fold cross validation technique.

Classifier	Accuracy		Precision		Recall		F1-score		Cohen_Kappa		MCC		G-Mean		AUC	
	Imb	Bal	Imb	Bal	Imb	Bal	Imb	Bal	Imb	Bal	Imb	Bal	Imb	Bal	Imb	Bal
Dataset 1																
Logistic Regression	0.6667	0.7500	0.6816	0.7476	0.6791	0.7421	0.6665	**0.7439**	0.3463	0.4884	0.3607	0.4897	0.6791	0.7421	0.6813	0.7216
Support Vector Machines	0.4881	0.5952	0.7313	0.6427	0.5426	0.6210	0.3947	**0.5868**	0.0757	0.2277	0.1984	0.2629	0.5426	0.6210	0.5970	0.6406
Decision Trees	0.8452	0.8571	0.8427	0.8551	0.8445	0.8551	0.8434	**0.8551**	0.6869	0.7102	0.6871	0.7102	0.8445	0.8551	0.8234	0.8324
Random Forest	0.8571	0.9167	0.8559	0.9267	0.8608	0.9083	0.8564	**0.9139**	0.7135	0.8285	0.7168	0.8348	0.8608	0.9083	0.8478	0.8777
Naive Bayes	0.6429	0.6548	0.7302	0.7368	0.6751	0.6857	0.6294	**0.6434**	0.3258	0.3466	0.4015	0.4194	0.6751	0.6857	0.6991	0.7083
K-Nearest Neighbor	0.5119	0.5952	0.5521	0.6000	0.5408	0.6009	0.4958	**0.5950**	0.0762	0.1973	0.0922	0.2009	0.5408	0.6009	0.5824	0.6188
Dataset 2																
Logistic Regression	0.9002	0.8626	0.7756	0.675001	0.5910	0.73239	0.6220	**0.6962**	0.2602	0.3950	0.3151	0.4030	0.5910	0.7324	0.2141	0.2788
Support Vector Machines	0.8757	0.8696	0.6542	0.6771	0.6168	0.7073	0.6099	**0.6902**	0.2353	0.3812	0.2555	0.3832	0.6168	0.7073	0.2071	0.2627
Decision Trees	0.8728	0.8204	0.6492	0.6079	0.6073	0.6534	0.6226	**0.6214**	0.2476	0.2488	0.2529	0.2570	0.6073	0.6534	0.1880	0.1903
Random Forest	0.8983	0.8437	0.7905	0.6605	0.5575	0.7499	0.5762	**0.6868**	0.1795	0.3808	0.2583	0.4002	0.5575	0.7499	0.1782	0.2744
Naive Bayes	0.8952	0.8848	0.7306	0.7159	0.7235	0.7773	0.7261	**0.7405**	0.4525	0.4826	0.4535	0.4894	0.7235	0.7773	0.3161	0.3448
K-Nearest Neighbor	0.9013	0.8465	0.7825	0.6781	0.5917	0.7971	0.6229	**0.7099**	0.2628	0.4299	0.3198	0.4599	0.5917	0.7971	0.2172	0.3175
Dataset 3																
Logistic Regression	0.7857	0.7857	0.7302	0.7386	0.7170	0.7712	0.7229	**0.7493**	0.4462	0.5016	0.4470	0.5087	0.7170	0.7712	0.4689	0.5047
Support Vector Machines	0.7381	0.7024	0.6578	0.6500	0.5759	0.6732	0.5754	**0.6555**	0.1873	0.3164	0.2189	0.3224	0.5759	0.6732	0.3351	0.3912
Decision Trees	0.9405	0.9524	0.9208	0.9315	0.9319	0.9537	0.9261	**0.9417**	0.8523	0.8834	0.8527	0.8849	0.9319	0.9537	0.8227	0.8536
Random Forest	0.9286	0.9524	0.9102	0.9315	0.9102	0.9537	0.9102	**0.9417**	0.8204	0.8834	0.8204	0.8849	0.9102	0.9537	0.7919	0.8536
Naive Bayes	0.7857	0.8095	0.7302	0.7605	0.7170	0.7605	0.7229	**0.7605**	0.4462	0.5210	0.4470	0.5210	0.7170	0.7605	0.4689	0.5206
K-Nearest Neighbor	0.7262	0.6190	0.6218	0.6245	0.5406	0.6565	0.5207	**0.6007**	0.1056	0.2462	0.1407	0.2791	0.5406	0.6565	0.3033	0.3636
Dataset 4																
Logistic Regression	0.6905	0.6071	0.8415	0.6288	0.5357	0.6429	0.4725	**0.6026**	0.0930	0.2443	0.2209	0.2713	0.5357	0.6429	0.3810	0.4184
Support Vector Machines	0.6905	0.5595	0.8415	0.6696	0.5357	0.6518	0.4725	**0.5580**	0.0930	0.2345	0.2209	0.3209	0.5357	0.6518	0.3810	0.4196
Decision Trees	0.8810	0.9167	0.8661	0.9291	0.8661	0.8839	0.8661	**0.9015**	0.7321	0.8037	0.7321	0.8118	0.8661	0.8839	0.7343	0.8230
Random Forest	0.8810	0.9167	0.9242	0.9291	0.8214	0.8839	0.8503	**0.9015**	0.7059	0.8037	0.7385	0.8118	0.8214	0.8839	0.7619	0.8230
Naive Bayes	0.7143	0.6786	0.6857	0.6551	0.6161	0.6696	0.6190	**0.6571**	0.2653	0.3193	0.2936	0.3244	0.6161	0.6696	0.4328	0.4497
K-Nearest Neighbor	0.6786	0.6667	0.6418	0.6207	0.5357	0.6161	0.4909	**0.6179**	0.0899	0.2364	0.1423	0.2367	0.5357	0.6161	0.3619	0.4107
Mean value of the measures to all the dataset																
Logistic Regression	0.7608	0.7514	0.7572	0.6975	0.6307	0.7221	0.6209	**0.6980**	0.2864	0.4073	0.3359	0.4182	0.6307	0.7221	0.4363	0.4809
Support Vector Machines	0.6981	0.6817	0.7212	0.6599	0.5677	0.6633	0.5131	**0.6226**	0.1478	0.2899	0.2234	0.3223	0.5677	0.6633	0.3800	0.4285
Decision Trees	0.8849	0.8867	0.8197	0.8309	0.8124	0.8365	0.8146	**0.8299**	0.6297	0.6615	0.6312	0.6660	0.8124	0.8365	0.6421	0.6748
Random Forest	0.8912	0.9074	0.8702	0.8619	0.7875	0.8739	0.7983	**0.8610**	0.6048	0.7241	0.6335	0.7329	0.7875	0.8739	0.6450	0.7072
Naive Bayes	0.7595	0.7569	0.7192	0.7171	0.6829	0.7233	0.6744	**0.7004**	0.3724	0.4174	0.3989	0.4386	0.6829	0.7233	0.4792	0.5058
K-Nearest Neighbor	0.7045	0.6819	0.6495	0.6308	0.5522	0.6676	0.5326	**0.6309**	0.1336	0.2775	0.1737	0.2942	0.5522	0.6676	0.3662	0.4277

Table A.13
Comparison Results between Proposed method and other included imbalanced learning approaches using all six learning techniques by employing ten fold cross validation technique.

Dataset	Performance measure	*CBReT* (LR)	SMOTE (LR)	SMOTEN (LR)	SVM-S (LR)	ADASYN (LR)	ROS (LR)	*CBReT* (KNN)	SMOTE (KNN)	SMOTEN (KNN)	SVM-S (KNN)	ADASYN (KNN)	ROS (KNN)
	precision	0.7476	0.7223	0.7298	0.7223	0.7257	0.7257	0.6000	0.5755	0.5529	0.5833	0.5651	0.5651
	recall	0.7421	0.7230	0.7316	0.7230	0.7271	0.7271	0.6009	0.5732	0.5489	0.5770	0.5655	0.5655
Dataset 1	f1-score	0.7439	0.7212	0.7297	0.7212	0.7250	0.7250	0.5950	0.5594	0.5258	0.5565	0.5599	0.5599
	Cohen_Kappa	0.4884	0.4436	0.4602	0.4436	0.4510	0.4510	0.1973	0.1413	0.0934	0.1473	0.1281	0.1281
	MCC	0.4897	0.4453	0.4614	0.4453	0.4527	0.4527	0.2009	0.1487	0.1017	0.1601	0.1306	0.1306
	precision	0.6750	0.6416	0.6245	0.6359	0.6222	0.6222	0.6781	0.6358	0.6535	0.6397	0.6337	0.6396
	recall	0.7324	0.7141	0.6134	0.7244	0.7146	0.7146	0.7971	0.7815	0.6371	0.7471	0.7676	0.7851
Dataset 2	f1-score	0.6962	0.6639	0.6185	0.6586	0.6416	0.6416	0.7099	0.6517	0.6445	0.6633	0.6516	0.6578
	Cohen_Kappa	0.3950	0.3343	0.2372	0.3279	0.2983	0.2983	0.4299	0.3341	0.2894	0.3413	0.3296	0.3438
	MCC	0.4030	0.3483	0.2377	0.3492	0.3239	0.3239	0.4599	0.3910	0.2901	0.3716	0.3783	0.3990
	precision	0.7386	0.7167	0.7195	0.7167	0.7122	0.7079	0.6245	0.5951	0.5500	0.5828	0.5916	0.5802
	recall	0.7712	0.7224	0.7359	0.7224	0.7413	0.7466	0.6565	0.5837	0.5349	0.5756	0.5891	0.5862
Dataset 3	f1-score	0.7493	0.7194	0.7263	0.7194	0.7215	0.7162	0.6007	0.5873	0.5321	0.5780	0.5902	0.5822
	Cohen_Kappa	0.5016	0.4388	0.4534	0.4388	0.4463	0.4396	0.2462	0.1771	0.0795	0.1575	0.1806	0.1658
	MCC	0.5087	0.4390	0.4551	0.4390	0.4526	0.4529	0.2791	0.1785	0.0836	0.1582	0.1807	0.1664
	precision	0.6288	0.5852	0.5641	0.6167	0.5893	0.5805	0.6207	0.5878	0.5635	0.5778	0.5650	0.5650
	recall	0.6429	0.5893	0.5714	0.6071	0.5893	0.5893	0.6161	0.5893	0.5536	0.5804	0.5625	0.5625
Dataset 4	f1-score	0.6026	0.5351	0.5411	0.5227	0.5238	0.5544	0.6179	0.5884	0.5534	0.5787	0.5634	0.5634
	Cohen_Kappa	0.2443	0.1460	0.1231	0.1667	0.1429	0.1527	0.2364	0.1770	0.1143	0.1579	0.1273	0.1273
	MCC	0.2713	0.1744	0.1353	0.2236	0.1786	0.1696	0.2367	0.1771	0.1166	0.1581	0.1275	0.1275
Dataset	Performance Measure	*CBReT* (SVM)	SMOTE (SVM)	SMOTEN (SVM)	SVM-S (SVM)	ADASYN (SVM)	ROS (SVM)	*CBReT* (DT)	SMOTE (DT)	SMOTEN (DT)	SVM-S (DT)	ADASYN (DT)	ROS (DT)
	precision	0.6427	0.6779	0.6865	0.6779	0.6864	0.6693	0.8551	0.8164	0.8313	0.8164	0.8089	0.8032
	recall	0.6210	0.5905	0.5910	0.5905	0.6070	0.5853	0.8551	0.8161	0.8340	0.8161	0.8106	0.8055
Dataset 1	f1-score	0.5868	0.5287	0.5294	0.5287	0.5583	0.5271	0.8551	0.8137	0.8299	0.8137	0.8069	0.8022
	Cohen_Kappa	0.2277	0.1782	0.1793	0.1782	0.2146	0.1715	0.7102	0.6287	0.6609	0.6287	0.6153	0.6055
	MCC	0.2629	0.2425	0.2511	0.2425	0.2667	0.2290	0.7102	0.6325	0.6653	0.6325	0.6195	0.6087

(*continued on next page*)

Table A.13 (*continued*).

	precision	0.6771	0.6081	0.6453	0.6772	0.6262	0.6321	0.6079	0.6082	0.6219	0.6082	0.5895	0.6100
	recall	0.7073	0.7139	0.6271	0.6834	0.6980	0.6739	0.6534	0.6212	0.5907	0.6212	0.6178	0.6288
Dataset 2	f1-score	0.6902	0.6194	0.6352	0.6802	0.6462	0.6477	0.6214	0.6140	0.6024	0.6140	0.5995	0.6180
	Cohen_Kappa	0.3812	0.2664	0.2708	0.3605	0.3010	0.2981	0.2488	0.2284	0.2069	0.2284	0.2019	0.2368
	MCC	0.3832	0.3041	0.2717	0.3606	0.3162	0.3032	0.2570	0.2290	0.2103	0.2290	0.2054	0.2380
	precision	0.6500	0.6566	0.6578	0.6566	0.6941	0.6941	0.9315	0.9102	0.9102	0.9208	0.9208	0.9208
	recall	0.6732	0.6030	0.5759	0.6030	0.5841	0.5841	0.9537	0.9102	0.9102	0.9319	0.9319	0.9319
Dataset 3	f1-score	0.6555	0.6111	0.5754	0.6111	0.5846	0.5846	0.9417	0.9102	0.9102	0.9261	0.9261	0.9261
	Cohen_Kappa	0.3164	0.2383	0.1873	0.2383	0.2111	0.2111	0.8834	0.8204	0.8204	0.8523	0.8523	0.8523
	MCC	0.3224	0.2540	0.2189	0.2540	0.2555	0.2555	0.8849	0.8204	0.8204	0.8527	0.8527	0.8527
	precision	0.6696	0.5758	0.5080	0.5758	0.6167	0.5952	0.9291	0.8661	0.8692	0.8661	0.8714	0.8714
	recall	0.6518	0.5446	0.5089	0.5446	0.6071	0.5804	0.8839	0.8661	0.8304	0.8661	0.8571	0.8571
Dataset 4	f1-score	0.5580	0.5322	0.5009	0.5322	0.5227	0.4845	0.9015	0.8661	0.8451	0.8661	0.8635	0.8635
	Cohen_Kappa	0.2345	0.1031	0.0164	0.1031	0.1667	0.1224	0.8037	0.7321	0.6916	0.7321	0.7273	0.7273
	MCC	0.3209	0.1164	0.0169	0.1164	0.2236	0.1750	0.8118	0.7321	0.6985	0.7321	0.7284	0.7284
Dataset	Performance Measure	*CBReT* (RF)	SMOTE (RF)	SMOTEN (RF)	SVM-S (RF)	ADASYN (RF)	ROS (RF)	*CBReT* (NB)	SMOTE (NB)	SMOTEN (NB)	SVM-S (NB)	ADASYN (NB)	ROS (NB)
	precision	0.9267	0.8649	0.8854	0.8649	0.8645	0.8706	0.7368	0.6951	0.6706	0.6951	0.6933	0.6933
	recall	0.9083	0.8634	0.8874	0.8634	0.8631	0.8696	0.6857	0.6601	0.6449	0.6601	0.6598	0.6598
Dataset 1	f1-score	0.9139	0.8635	0.8857	0.8635	0.8634	0.8696	0.6434	0.6229	0.6113	0.6229	0.6232	0.6232
	Cohen_Kappa	0.8285	0.7272	0.7717	0.7272	0.7269	0.7393	0.3466	0.3001	0.2727	0.3001	0.2997	0.2997
	MCC	0.8348	0.7283	0.7728	0.7283	0.7276	0.7402	0.4194	0.3534	0.3144	0.3534	0.3515	0.3515
	precision	0.6605	0.6907	0.7349	0.6907	0.6869	0.6928	0.7159	0.7001	0.4997	0.7075	0.7019	0.7019
	recall	0.7499	0.6456	0.5363	0.6456	0.6532	0.6544	0.7773	0.7310	0.4993	0.7498	0.7561	0.7561
Dataset 2	f1-score	0.6868	0.6636	0.5418	0.6636	0.6675	0.6704	0.7405	0.7138	0.3484	0.7255	0.7238	0.7238
	Cohen_Kappa	0.3808	0.3287	0.1168	0.3287	0.3358	0.3418	0.4826	0.4281	−0.001	0.4519	0.4491	0.4491
	MCC	0.4002	0.3333	0.1848	0.3333	0.3384	0.3450	0.4894	0.4300	−0.001	0.4553	0.4548	0.4548
	precision	0.9315	0.9304	0.9208	0.9304	0.9145	0.9208	0.7607	0.7302	0.7605	0.7456	0.7456	0.7456
	recall	0.9537	0.9184	0.9319	0.9184	0.9455	0.9319	0.7741	0.7170	0.7605	0.7388	0.7388	0.7388
Dataset 3	f1-score	0.9417	0.9241	0.9261	0.9241	0.9280	0.9261	0.7667	0.7229	0.7605	0.7420	0.7420	0.7420
	Cohen_Kappa	0.8834	0.8483	0.8523	0.8483	0.8562	0.8523	0.5337	0.4462	0.5210	0.4841	0.4841	0.4841
	MCC	0.8849	0.8487	0.8527	0.8487	0.8594	0.8527	0.5346	0.4470	0.5210	0.4843	0.4843	0.4843
	precision	0.9291	0.8692	0.8714	0.8692	0.8607	0.8892	0.6551	0.5557	0.6143	0.5557	0.6197	0.5400
	recall	0.8839	0.8304	0.8571	0.8304	0.8393	0.8661	0.6696	0.5625	0.5714	0.5625	0.6339	0.5446
Dataset 4	f1-score	0.9015	0.8451	0.8635	0.8451	0.8484	0.8760	0.6571	0.5382	0.4367	0.5382	0.6003	0.5170
	Cohen_Kappa	0.8037	0.6916	0.7273	0.6916	0.6972	0.7523	0.3193	0.1094	0.1039	0.1094	0.2326	0.0769
	MCC	0.8118	0.6985	0.7284	0.6985	0.6996	0.7549	0.3244	0.1180	0.1807	0.1180	0.2532	0.0846

Table A.14
Comparison Results based on the G-mean and AUC between Proposed method and other imbalanced learning approaches using all six learning techniques by employing ten fold cross validation technique.

Dataset	Performance Measure	*CBReT* (LR)	SMOTE (LR)	SMOTEN (LR)	SVM-S (LR)	ADASYN (LR)	ROS (LR)	*CBReT* (KNN)	SMOTE (KNN)	SMOTEN (KNN)	SVM-S (KNN)	ADASYN (KNN)	ROS (KNN)
Dataset 1	G-Mean	**0.7421**	0.7230	0.7316	0.7230	0.7271	0.7271	0.6009	**0.5732**	0.5489	0.5770	0.5655	0.5655
	AUC	**0.7216**	0.7107	0.7179	0.7107	0.7142	0.7042	0.6188	**0.6020**	0.5878	0.6051	0.5964	0.5964
Dataset 2	G-Mean	**0.7324**	0.7141	0.6134	0.7244	0.7146	0.7146	0.7971	**0.7815**	0.6371	0.7471	0.7676	0.7851
	AUC	**0.2788**	0.2390	0.1761	0.2391	0.2235	0.2235	0.3175	**0.2616**	0.2031	0.2526	0.2545	0.2673
Dataset 3	G-Mean	**0.7712**	0.7224	0.7359	0.7224	0.7413	0.7466	0.6565	**0.5837**	0.5349	0.5756	0.5891	0.5862
	AUC	**0.5047**	0.4622	0.4715	0.4622	0.4671	0.4641	0.3636	**0.3250**	0.2919	0.3177	0.3267	0.3220
Dataset 4	G-Mean	**0.6429**	0.5893	0.5714	0.6071	0.5893	0.5893	0.6161	**0.5893**	0.5536	0.5804	0.5625	0.5625
	AUC	**0.4184**	0.3805	0.3706	0.3905	0.3801	0.3815	0.4107	**0.3867**	0.3639	0.3798	0.3686	0.3686
Dataset	Performance Measure	*CBReT* (LR)	SMOTE (SVM)	SMOTEN (SVM)	SVM-S (SVM)	ADASYN (SVM)	ROS (SVM)	*CBReT* (DT)	SMOTE (DT)	SMOTEN (DT)	SVM-S (DT)	ADASYN (DT)	ROS (DT)
Dataset 1	G-Mean	**0.6210**	0.5905	0.5910	0.5905	0.6070	0.5853	0.8551	**0.8161**	0.8340	0.8161	0.8106	0.8055
	AUC	**0.6406**	0.6175	0.6189	0.6175	0.6262	0.6124	0.8324	**0.7973**	0.8189	0.7973	0.7946	0.7884
Dataset 2	G-Mean	0.7073	**0.7139**	0.6271	0.6834	0.6980	0.6739	0.6534	**0.6212**	0.5907	0.6212	0.6178	0.6288
	AUC	**0.2627**	0.2112	0.1930	0.2470	0.2195	0.2120	0.1903	**0.1731**	0.1626	0.1731	0.1635	0.1776
Dataset 3	G-Mean	**0.6732**	0.6030	0.5759	0.6030	0.5841	0.5841	0.9537	**0.9102**	0.9102	0.9319	0.9319	0.9319
	AUC	**0.3912**	0.3544	0.3351	0.3544	0.3502	0.3502	0.8536	**0.7919**	0.7919	0.8227	0.8227	0.8227
Dataset 4	G-Mean	**0.6518**	0.5446	0.5089	0.5446	0.6071	0.5804	0.8839	**0.8661**	0.8304	0.8661	0.8571	0.8571
	AUC	**0.4196**	0.3608	0.3374	0.3608	0.3905	0.3741	0.8230	**0.7343**	0.7164	0.7343	0.7363	0.7363
Dataset	Performance Measure	*CBReT* (LR)	SMOTE (RF)	SMOTEN (RF)	SVM-S (RF)	ADASYN (RF)	ROS (RF)	*CBReT* (NB)	SMOTE (NB)	SMOTEN (NB)	SVM-S (NB)	ADASYN (NB)	ROS (NB)
Dataset 1	G-Mean	0.9083	0.8634	0.8874	0.8634	0.8631	0.8696	0.6857	**0.6601**	0.6449	0.6601	0.6598	0.6598
	AUC	0.8777	0.8411	0.8710	0.8411	0.8404	0.8478	0.7083	**0.6785**	0.6619	0.6785	0.6778	0.6778
Dataset 2	G-Mean	0.7499	0.6456	0.5363	0.6456	0.6532	0.6544	0.7773	**0.7310**	0.4993	0.7498	0.7561	0.7561
	AUC	0.2744	0.2280	0.1457	0.2280	0.2316	0.2357	0.3448	**0.2971**	0.1086	0.3169	0.3162	0.3162
Dataset 3	G-Mean	0.9537	0.9184	0.9319	0.9184	0.9455	0.9319	0.7741	**0.7170**	0.7605	0.7388	0.7388	0.7388
	AUC	0.8536	0.8262	0.8227	0.8262	0.8213	0.8227	0.5116	0.4689	**0.5206**	0.4945	0.4945	0.4945
Dataset 4	G-Mean	0.8839	0.8304	0.8571	0.8304	0.8393	0.8661	0.6696	**0.5625**	0.5714	0.5625	0.6339	0.5446
	AUC	0.8230	0.7164	0.7363	0.7164	0.7133	0.7629	0.4497	**0.3655**	0.3687	0.3655	0.4127	0.3553

References

[1] A. Yamashita, S. Counsell, Code smells as system-level indicators of maintainability: An empirical study, J. Syst. Softw. 86 (10) (2013) 2639–2653.

[2] P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162), Vol. 6, No. 4, Dagstuhl Reports, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[3] F. Khomh, M.D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change-and fault-proneness, Empir. Softw. Eng. 17 (2012) 243–275.

[4] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, A. Bacchelli, On the relation of test smells to software code quality, in: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2018, pp. 1–12.

[5] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, E. Figueiredo, A review-based comparative study of bad smell detection tools, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, 2016, pp. 1–12.

[6] E. Murphy-Hill, A.P. Black, An interactive ambient visualization for code smells, in: Proceedings of the 5th International Symposium on Software Visualization, 2010, pp. 5–14.

[7] A. Kaur, S. Jain, S. Goel, A support vector machine based approach for code smell detection, in: 2017 International Conference on Machine Learning and Data Science, MLDS, IEEE, 2017, pp. 9–14.

[8] R.S. Rao, S. Dewangan, A. Mishra, M. Gupta, A study of dealing class imbalance problem with machine learning methods for code smell severity detection using PCA-based feature selection technique, Sci. Rep. 13 (1) (2023) 16245.

[9] A. Alazba, H. Aljamaan, Code smell detection using feature selection and stacking ensemble: An empirical investigation, Inf. Softw. Technol. 138 (2021) 106648.

[10] S. Wang, X. Yao, Using class imbalance learning for software defect prediction, IEEE Trans. Reliab. 62 (2) (2013) 434–443.

[11] Y. Zhang, C. Dong, MARS: Detecting brain class/method code smell based on metric–attention mechanism and residual network, J. Softw.: Evol. Process (2021) e2403.

[12] F. Pecorelli, D. Di Nucci, C. De Roover, 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, 2019, pp. 19–24.

[13] S. Dewangan, R.S. Rao, A. Mishra, M. Gupta, Code smell detection using ensemble machine learning algorithms, Appl. Sci. 12 (20) (2022) 10321.

[14] F. Arcelli Fontana, M.V. Mäntylä, M. Zanoni, A. Marino, Comparing and experimenting machine learning techniques for code smell detection, Empir. Softw. Eng. 21 (3) (2016) 1143–1191.

[15] K. Beck, M. Fowler, G. Beck, Bad smells in code, Refact.: Improv. Des. Exist. Code 1 (1999) (1999) 75–88.

[16] O. Ciupke, Automatic detection of design problems in object-oriented reengineering, in: Proceedings of Technology of Object-Oriented Languages and Systems-TOOLS, Vol. 30, Cat. No. PR00278, IEEE, 1999, pp. 18–32.

[17] R. Marticorena, C. López, Y. Crespo, Parallel inheritance hierarchy: Detection from a static view of the system, in: 6th International Workshop on Object Oriented Reenginering, WOOR, Glasgow, UK, 2005, p. 6.

[18] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, Jdeodorant: Identification and removal of feature envy bad smells, in: 2007 IEEE International Conference on Software Maintenance, IEEE, 2007, pp. 519–520.

[19] F. Pecorelli, F. Palomba, D. Di Nucci, A. De Lucia, Comparing heuristic and machine learning approaches for metric-based code smell detection, in: 2019 IEEE/ACM 27th International Conference on Program Comprehension, ICPC, IEEE, 2019, pp. 93–104.

[20] N. Maneerat, P. Muenchaisri, Bad-smell prediction from software design model using machine learning techniques, in: 2011 Eighth International Joint Conference on Computer Science and Software Engineering, JCSSE, IEEE, 2011, pp. 331–336.

[21] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, E. Aimeur, Support vector machines for anti-pattern detection, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012, pp. 278–281.

[22] T. Menzies, A. Marcus, Automated severity assessment of software defect reports, in: 2008 IEEE International Conference on Software Maintenance, IEEE, 2008, pp. 346–355.

[23] N. Junsomboon, T. Phienthrakul, Combining over-sampling and under-sampling techniques for imbalance dataset, in: Proceedings of the 9th International Conference on Machine Learning and Computing, 2017, pp. 243–247.

[24] W. Zhang, R. Ramezani, A. Naeim, WOTBoost: Weighted oversampling technique in boosting for imbalanced learning, in: 2019 IEEE International Conference on Big Data, Big Data, IEEE, 2019, pp. 2523–2531.

[25] J. Li, Q. Zhu, Q. Wu, Z. Fan, A novel oversampling technique for class-imbalanced learning based on SMOTE and natural neighbors, Inform. Sci. 565 (2021) 438–455.

[26] M.H. Ibrahim, ODBOT: Outlier detection-based oversampling technique for imbalanced datasets learning, Neural Comput. Appl. 33 (22) (2021) 15781–15806.

[27] A. Islam, S.B. Belhaouari, A.U. Rehman, H. Bensmail, KNNOR: An oversampling technique for imbalanced datasets, Appl. Soft Comput. 115 (2022) 108288.

[28] X. Tao, W. Chen, X. Zhang, W. Guo, L. Qi, Z. Fan, SVDD boundary and DPC clustering technique-based oversampling approach for handling imbalanced and overlapped data, Knowl.-Based Syst. 234 (2021) 107588.

[29] J. Wei, H. Huang, L. Yao, Y. Hu, Q. Fan, D. Huang, New imbalanced fault diagnosis framework based on cluster-MWMOTE and MFO-optimized LS-SVM using limited and complex bearing data, Eng. Appl. Artif. Intell. 96 (2020) 103966.

[30] J. Wei, H. Huang, L. Yao, Y. Hu, Q. Fan, D. Huang, NI-MWMOTE: An improving noise-immunity majority weighted minority oversampling technique for imbalanced classification problems, Expert Syst. Appl. 158 (2020) 113504.

[31] J. Wei, H. Huang, L. Yao, Y. Hu, Q. Fan, D. Huang, IA-SUWO: An Improving Adaptive semi-unsupervised weighted oversampling for imbalanced classification problems, Knowl.-Based Syst. 203 (2020) 106116.

[32] M. De Stefano, F. Pecorelli, F. Palomba, A. De Lucia, Comparing within-and cross-project machine learning algorithms for code smell detection, in: Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution, 2021, pp. 1–6.

[33] Y. Zhang, C. Ge, S. Hong, R. Tian, C. Dong, J. Liu, DeleSmell: Code smell detection based on deep learning and latent semantic analysis, Knowl.-Based Syst. 255 (2022) 109737.

[34] J. Nanda, J.K. Chhabra, SSHM: SMOTE-stacked hybrid model for improving severity classification of code smell, Int. J. Inf. Technol. 14 (5) (2022) 2701–2707.

[35] S. Boutaib, S. Bechikh, F. Palomba, M. Elarbi, M. Makhlouf, L.B. Said, Code smell detection and identification in imbalanced environments, Expert Syst. Appl. 166 (2021) 114076.

[36] S.S. Chouhan, S.S. Rathore, Generative adversarial networks-based imbalance learning in software aging-related bug prediction, IEEE Trans. Reliab. 70 (2) (2021) 626–642.

[37] P. Bholowalia, A. Kumar, EBK-means: A clustering technique based on elbow method and k-means in WSN, Int. J. Comput. Appl. 105 (9) (2014).

[38] H. Grodzicka, A. Ziobrowski, Z. Łakomiak, M. Kawa, L. Madeyski, Code smell prediction employing machine learning meets emerging java language constructs, in: Data-Centric Business and Applications: Towards Software Development, Vol. 4, Springer, 2020, pp. 137–167.

[39] S. Dewangan, R.S. Rao, S.R. Chowdhuri, M. Gupta, Severity classification of code smells using machine-learning methods, SN Comput. Sci. 4 (5) (2023) 564.

[40] S. Dewangan, R.S. Rao, A. Mishra, M. Gupta, A novel approach for code smell detection: an empirical study, IEEE Access 9 (2021) 162869–162883.

[41] A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, G. Sladić, Automatic detection of Long Method and God Class code smells through neural source code embeddings, Expert Syst. Appl. 204 (2022) 117607.

[42] M. Galar, A. Fernandez, E. Barrenechea, H. Bustince, F. Herrera, A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches, IEEE Trans. Syst., Man, Cybern., C (Appl. Rev.) 42 (4) (2011) 463–484.

[43] M. Bekkar, H.K. Djemaa, T.A. Alitouche, Evaluation measures for models assessment over imbalanced data sets, J. Inf. Eng. Appl. 3 (10) (2013).

[44] C.G. Weng, J. Poon, A new evaluation measure for imbalanced datasets, in: Proceedings of the 7th Australasian Data Mining Conference, Vol. 87, 2008, pp. 27–32.

[45] D. Chicco, Ten quick tips for machine learning in computational biology, BioData Min. 10 (1) (2017) 1–17.

[46] R.F. Woolson, Wilcoxon signed-rank test, in: Wiley Encyclopedia of Clinical Trials, Wiley Online Library, 2007, pp. 1–3.

[47] S. Jain, A. Saha, Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection, Sci. Comput. Programm. 212 (2021) 102713.

[48] N. Pritam, M. Khari, R. Kumar, S. Jha, I. Priyadarshini, M. Abdel-Basset, H.V. Long, et al., Assessment of code smell for predicting class change proneness using machine learning, IEEE Access 7 (2019) 37414–37425.

[49] S.R. Safavian, D. Landgrebe, A survey of decision tree classifier methodology, IEEE Trans. Syst. Man Cybern. 21 (3) (1991) 660–674.

[50] A. Liaw, M. Wiener, et al., Classification and regression by randomForest, R News 2 (3) (2002) 18–22.

[51] D.W. Hosmer Jr., S. Lemeshow, R.X. Sturdivant, Applied Logistic Regression, Vol. 398, John Wiley & Sons, 2013.

[52] K.P. Murphy, et al., Naive bayes classifiers, Univ. British Columbia 18 (60) (2006) 1–8.

[53] S.S. Keerthi, S.K. Shevade, C. Bhattacharyya, K.R. Murthy, A fast iterative nearest point algorithm for support vector machine classifier design, IEEE Trans. Neural Netw. 11 (1) (2000) 124–136.

[54] S. Zhang, X. Li, M. Zong, X. Zhu, D. Cheng, Learning k for knn classification, ACM Trans. Intell. Syst. Technol. 8 (3) (2017) 1–19.

[55] N.A.A. Khleel, K. Nehéz, Deep convolutional neural network model for bad code smells detection based on oversampling method, Indonesian J. Electr. Eng. Comput. Sci. 26 (3) (2022) 1725–1735.

[56] F. Li, K. Zou, J.W. Keung, X. Yu, S. Feng, Y. Xiao, On the relative value of imbalanced learning for code smell detection, Softw. - Pract. Exp. (2023).

[57] H. Gupta, S. Misra, L. Kumar, N.B. Murthy, An empirical study to investigate data sampling techniques for improving code-smell prediction using imbalanced data, in: Information and Communication Technology and Applications: Third International Conference, ICTA 2020, Minna, Nigeria, November 24–27, 2020, Revised Selected Papers 3, Springer, 2021, pp. 220–233.

[58] T. Lin, X. Fu, F. Chen, L. Li, A novel approach for code smells detection based on deep leaning, in: Applied Cryptography in Computer and Communications: First EAI International Conference, AC3 2021, Virtual Event, May 15-16, 2021, Proceedings 1, Springer, 2021, pp. 171–174.

[59] F. Li, K. Zou, J.W. Keung, X. Yu, S. Feng, Y. Xiao, On the relative value of imbalanced learning for code smell detection, Softw. - Pract. Exp. 53 (10) (2023) 1902–1927, http://dx.doi.org/10.1002/spe.3235, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3235.

[60] R. Sandouka, H. Aljamaan, Python code smells detection using conventional machine learning models, PeerJ Comput. Sci. 9 (2023) e1370.

[61] T. Sharma, V. Efstathiou, P. Louridas, D. Spinellis, Code smell detection by deep direct-learning and transfer-learning, J. Syst. Softw. 176 (2021) http://dx.doi.org/10.1016/j.jss.2021.110936.