A Clustering-Based Resampling Technique with Cluster Structure Analysis for Software Defect Detection in Imbalanced Datasets

Leonidas Akritidis[a,*], Panayiotis Bozanis[a]

[a]School of Science and Technology, International Hellenic University, 14th km Thessaloniki – N. Moudania, Thessaloniki, Greece

Abstract

Software defect detection focuses on the automatic identification of flaws in software modules. Given the great importance of the problem, numerous researchers have introduced a rich collection of deep learning approaches to confront it. However, the datasets that are used to train the proposed classifiers are in most cases highly imbalanced, leading to models that cannot learn the minority classes effectively, while being biased towards the majority class. The state-of-the-art solutions either overlook the issue of data imbalance, or they confront it insufficiently by ignoring the existence of outliers and the local properties of the classes' distributions. In this work we introduce CBR, a Clustering-Based Resampling technique for mitigating the problem of class imbalance in software defect detection tasks. The proposed method initially employs a quite simple heuristic to determine the maximum distance threshold between two clusters. Then, it uses this threshold to apply hierarchical clustering with the aim of grouping together similar samples. CBR considers the singleton clusters as outliers, and discards the ones originating from the majority class. The algorithm subsequently organizes the clusters into sub-clusters than contain samples from the same class and determines which sub-clusters should participate in the oversampling process. In this way, CBR produces samples of improved quality and variance. We evaluated the performance of CBR against 9 baseline and state-of-the-art techniques by using 27 datasets and a Multilayer Perceptron classifier. The results demonstrate the superiority of CBR in terms of Balanced Accuracy and Precision scores.

Keywords: software defect detection, oversampling, imbalanced data, classification, clustering, CBR

1. Introduction

Software development is a challenging procedure, usually involving multiple phases. From the initial understanding of the project requirements, until the final product release, significant amounts of time and resources are consumed. Among these phases, the processes of code debugging and beta testing affect the overall quality in a critical manner, since they determine user experience, productivity, and efficiency. Hence, the automatic detection of software defects plays an important role in all these aspects, whereas it also contributes to the limitation of the development costs.

As the size and complexity of modern software grows constantly, many researchers employed machine learning solutions to automatically detect flaws in software modules. Examples include well-known classifiers like Logistic Regression [3], Support Vector Machines (SVMs) [10], Tree-based learners [15, 43], fully-connected Neural Networks [21, 42], Convolutional Neural Networks (CNNs) [25], Long-Short Term Memory (LSTM) units [26, 31], and so forth.

Despite the robustness of the aforementioned models, the vast majority of datasets on which they are trained are highly imbalanced. Class imbalance occurs when the dataset samples are unevenly distributed to the involved classes. Large imbalance ratios lead to bad classification performance, because the models are trained with problematic data. Hence, they tend to be biased towards the majority class and they cannot reliably learn the latent properties of the minority classes.

[*]Corresponding author

Email addresses: lakritidis@ihu.gr (Leonidas Akritidis[a,]), pbozanis@ihu.gr (Panayiotis Bozanis[a])

The relevant literature contains numerous solutions for alleviating class imbalance. Two of the most widespread techniques are *oversampling* and *undersampling*. The first one restores balance by generating artificial data samples belonging to the minority class, whereas the second one removes samples from the majority class. However, most traditional oversampling approaches synthesize samples between two neighbors without examining neither their type, nor their absolute distances. Consequently, they tend to generate noisy samples (e.g., outliers between other outliers) that further degrade classification effectiveness. Moreover, finding the k-nearest neighbors is an expensive process, especially when the dataset is large and highly dimensional. These weaknesses are common in the Synthetic Minority Oversampling Technique (SMOTE) [6], several of its numerous variants [11], Adaptive Synthetic Sampling (ADASYN) [20], and other methods.

On the other hand, the clustering-based oversampling techniques like k-Means SMOTE [9] do not analyze thoroughly the structure of the generated clusters, and synthesize samples merely by using SMOTE. Therefore, they adopt all the aforementioned drawbacks. In addition, they typically employ the original k-Means algorithm that does not identify the outliers.

Another family of oversampling methods includes deep generative models like Generative Adversarial Networks (GANs) [16], Variational Autoencoders (VAEs) [23], etc. Although these models have exhibited impressive performance in several data generation tasks, they require a lot of data for training, and this is not always possible in software defect detection tasks. Furthermore, their training is occasionally unstable (due to effects like mode collapse, vanishing gradients, or other phenomena), requires extensive fine-tuning, and it is much more expensive than the classical data mining approaches.

To overcome the aforementioned problems, in this article we propose a new, effective resampling technique called Cluster-Based Resampling (CBR). CBR mitigates the problem of class imbalance by constructing clusters of similar samples and analyzing their internal structure. In contrast to other solutions that employ k-Means, here we apply Hierarchical Agglomerative Clustering (HAC). Compared to k-Means, HAC does not require previous knowledge of the number of clusters whereas it can effectively detect outliers. Furthermore, CBR takes into consideration the possible existence of outliers in the dataset. It considers the clusters that remain singletons after the execution of HAC as anomalies and treats them according to their class. In this way, it improves both the quality and the diversity of the underlying dataset.

Additionally, we introduce Minority Centroid Oversampling (MCO), a simple yet effective technique that synthesizes data between a minority sample and the centroid of its respective (sub)cluster. In this wise, it avoids submitting expensive spatial queries and reduces the risk of creating samples too far away from the original ones. The following list summarizes the novel elements of CBR that constitute the contributions of our work.

- We introduce a novel resampling technique called Cluster-Based Resampling (CBR) that improves the software defect detection performance in imbalanced datasets.
- CBR augments the underlying data in the following ways: i) it uses HAC to create clusters of samples with similar properties, ii) it takes into consideration the possible existence of outliers, and iii) it analyzes the clusters' structure by forming sub-clusters of samples from the same class.
- The proposed method is the first to intimately perform both undersampling (by pruning the majority class outliers) and oversampling (by generating minority data instances to bring balance to a cluster).
- CBR does not require expensive spatial queries for locating nearest neighbors, or neighbors within a fixed radius. For this reason, its performance is particularly fast.
- We introduce a simple, yet effective heuristic to determine the maximum distance that two clusters must have in order to be merged by HAC. With this heuristic, CBR becomes a parameter-free algorithm.

The rest of the paper is organized as follows: Section 2 contains a literature overview of the area of software defect detection. Sections 3 and 4 describe the details of Cluster-Based Resampling and Minority Centroid Oversampling, respectively. The results of the experimental evaluation of the proposed techniques are presented and discussed in Section 5. Finally, the article is concluded with Section 6 that summarizes the findings of this work and highlights several key elements of our future research.

2. Related Work

The development of large-scale software projects has rendered the automatic detection of defects particularly important. Although a variety of techniques have been proposed so far, the problem is primarily confronted in the literature as a classification task [24]. In this context, numerous research groups have introduced high quality solutions that utilize state-of-the-art classifiers to improve the detection performance.

More specifically, in [36] the authors proposed a simple neural network composed of two fully-connected hidden layers. They also emphasized on data preprocessing by applying min-max normalization and log transformation. The work of Arar et al. introduced a shallow neural network with one hidden layer that was trained with the Artificial Bee Colony algorithm [2]. The architecture was attested on five datasets and outperformed 5 traditional classifiers.

More recently, Manjula et al. [28] focused on feature optimization and suggested two such techniques combined with a deep neural network. The first technique was a genetic algorithm, whereas the second one was an Adaptive Denoising Autoencoder for learning better feature representations. In [22], Jing et al. introduced a dictionary-based technique that learns multiple structures along with sparse representation coefficients. In addition, the authors adopted a cost-sensitive learning approach, arguing that the correct classification of defective modules is more important than the classification of healthy ones. Another cost-sensitive method with siamese neural networks was introduced in [48]. Finally, [33] presented a brief survey on deep learning tools for software defect detection.

Apart from neural networks, other researchers have experimented with different classifiers. Hence, Perreault et al. studied the performance of SVMs, Naive Bayes, Logistic Regression and k-Nearest Neighbors (k-NN) on 5 datasets [35]. In [18], an overview of Logistic Regression in defect detection tasks was conducted. Moreover, [34] examined four decision making approaches by utilizing 10 datasets, 38 classifiers, and 13 evaluation measures. They concluded that the best detection performance was obtained by applying boosting at the CART and C4.5 models.

The negative effects of class imbalance in classification performance led to the introduction of multiple techniques with the aim of mitigating the problem. In [6], Chawla et al. introduced the Synthetic Minority Oversampling Technique (SMOTE), a simple yet strong strategy for synthesizing artificial samples for the minority classes. Initially, the k nearest neighbors (say, $k = 5$) of a minority sample are computed. Subsequently, new data instances are created at random points over the line that connects the sample with its neighbor.

However, SMOTE generates synthetic samples between two neighbors without examining their type (e.g. outlier, core point, etc.), or their distance; this means that it may create outliers between other outliers. In addition, SMOTE performs multiple spatial searches (k-NN queries) that can be expensive, especially in highly dimensional datasets. These drawbacks are also present in other methods, like ADASYN [20] and Random Oversampling (ROS).

Because of its simplicity and good performance, SMOTE was extensively utilized in numerous applications involving imbalanced data [5, 14]. In addition, a large family of variants was introduced [11]. The most popular among them include Borderline-SMOTE [19], Safe-Level SMOTE [4], SMOTEBoost [7] and SMOTEBagging [44]. Galar et al. investigated numerous ensemble-based approaches, including boosting, bagging and hybrid methods [40].

Taking a different approach, several works of the relevant literature utilize clustering algorithms either for oversampling or for undersampling purposes. Regarding the former, the method of [9] initially executes k-Means to form clusters of similar elements. Then, it balances each cluster by utilizing SMOTE. As mentioned earlier, this algorithm does not take into account the existence of outliers, whereas it does not study thoroughly the clusters' structure. Clustering has also been applied to undersampling scenarios, where the representative point of each cluster (e.g., the centroid) is used to reduce the number of majority class samples [27, 47, 46].

Affected by the excellent performance of the recent deep generative models, an increasing number of authors utilized Generative Adversarial Networks (GANs) [16, 17] and Variational Autoencoders (VAEs) [23] to alleviate class imbalance. Specifically, the conditional GAN (cGAN) is capable of producing data instances belonging to a particular class [29]. Safe-Borderline GAB (sbGAN) extended cGAN by characterizing each samples as core, safe, isolated, or borderline [1]. Xu et al. took into consideration that tabular data may possess both continuous and discrete values [45]. They proposed ctGAN, a model that captures multiple modes of continuous variables and the imbalanced nature of the discrete ones.

Finally, the Capsule Network (CapsNet) is a deep learning model with remarkable performance in image classification tasks with imbalanced data [38]. This model is not generative and does not strictly fall into the category of the resampling methods. Although it has been used solely with image data, the recent work of Chen et al. demonstrated that it can be used effectively with tabular data too [8].

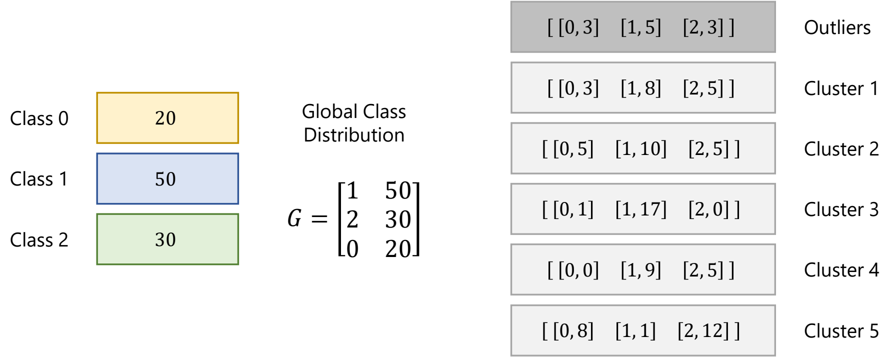

Figure 1: An imbalanced dataset with 100 samples and 3 classes. The matrix G stores the class distribution of the dataset. Its first row denotes the majority class. On the right part, a clustering algorithm identifies 12 outliers; the rest 88 samples are grouped within 5 clusters. Inside each light gray rectangle, the local class distribution matrices L are shown. For example, cluster 5 contains 8 samples from class 0, 1 sample from class 1 and 12 samples from class 2.

3. Cluster-Based Resampling

Let $(\mathbf{X}, Y) \rightarrow \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ be an imbalanced dataset with N samples, where $\mathbf{x}_i$ and y_i represent the input vector and the target variable of the i-th sample respectively. Cluster-Based Resampling (CBR) confronts the imbalance problem first by clustering the dataset elements, and then by balancing each cluster individually through a novel, unsupervised mechanism that analyzes the clusters' structure. Its output is an *augmented* dataset $(\mathbf{X}', Y')$ that may *not* necessarily be perfectly balanced.

In software defect detection tasks, the classes Y usually receive binary values that indicate defective/non-defective software modules. However, here we consider a more generic approach that allows the classes to receive any value from a set of $|Y|$ discrete values, enabling CBR to be applied at any multi-class problem.

The basic steps of CBR are presented in the pseudocode of Algorithm 1. Initially, the algorithm constructs a $|Y| \times 2$ global class distribution matrix G. Each row of G consists of two elements; a class label y, and the number of samples $|Y_y \subset Y|$ belonging to class y. G is subsequently sorted with respect to the values of its second column, so that the majority class is placed at the first row, the second most multitudinous class is placed in the second row, and so forth (steps 1–3). A representative example is described in Figure 1.

In the next phase, a clustering algorithm is applied to the samples of the dataset and assigns cluster labels to them. It is desirable that this algorithm possesses the following properties:

- *Uniqueness*: A data instance $\mathbf{x}_i$ must be placed into precisely one cluster.
- *Outlier detection*: Outliers are samples that do not comply with the probability distribution of their own class. Such cases are typically considered as noise that negatively affect the defect detection performance.
- *Independence from hyper-parameters*: Most clustering methods require tuning several hyper-parameters. However, setting their values in advance can be challenging, since the underlying dataset may be completely unknown. For example, k-Means requires prior knowledge of the number of clusters to be created.

In this work, we selected Hierarchical Agglomerative Clustering (HAC) because it fulfils the first two requirements by definition. The hierarchical clustering methods work in an either top-to-bottom (called divisive), or in a bottom-to-top (called agglomerative) fashion. HAC belongs to the second category. Initially, it places each sample into its own singleton cluster. Then, it progressively merges the two closest clusters, provided that their distance does not exceed a threshold t_d. The process stops when all clusters are separated by distances greater than t_d, so no cluster merging is possible any more.

HAC can be combined with several strategies for computing the distance $d(c_1, c_2)$ between two clusters c_1, c_2. For example, single linkage sets $d(c_1, c_2)$ equal to the distance between the closest pairs of elements from c_1 and c_2, respectively. The opposite approach is adopted by complete, or maximum linkage. In CBR we employed Ward, a variance-minimizing approach that sets $d(c_1, c_2)$ equal to the sum of squared differences within all clusters.

Algorithm 1 Cluster-Based Resampler

Input The imbalanced dataset $(\mathbf{X}, Y)$
Output The augmented dataset $(\mathbf{X}', Y')$

1: $G \leftarrow \Big[[y, |Y_y|]\Big], y \in Y$ ▷ *Global class distribution: $|Y| \times 2$ matrix that stores the number of samples per class*
2: $G \leftarrow$ sort G in decreasing $|Y_y|$ order
3: MajorityClass $\leftarrow G[0, 0]$ ▷ *The first item of sorted G reveals the majority class*
4: $C \leftarrow$ Agglomerative $(\mathbf{X}, Y, t_d = \tilde{d}/3)$ ▷ *Perform clustering on the dataset $(\mathbf{X}, Y)$ and obtain the cluster set C*
5: $(\mathbf{X}', Y') \leftarrow [\ ]$ ▷ *The augmented dataset to be returned*
6: IncClasses $\leftarrow 0$ ▷ *Number of classes to oversample*
7: **for** each cluster $c \in C$ **do** ▷ *Analyze the obtained clusters one by one*
8: $\quad\Big(\mathbf{X}^{(c)}, Y^{(c)}\Big) \leftarrow$ contents of c ▷ *The samples that have been placed in cluster c*
9: $\quad\Big(\mathbf{X}^{(c)}_{inc}, Y^{(c)}_{inc}\Big) \leftarrow [\ ]$ ▷ *The samples that will be included for oversampling: initially empty*
10: $\quad L \leftarrow \Big[[y, |Y^{(c)}_y|]\Big], y \in Y^{(c)}$ ▷ *In-cluster class distribution: similar to G (line 1), but in a per-cluster fashion*
11: $\quad$MaxClusterSamples $\leftarrow \max L[:, 1]$ ▷ *Maximum number of samples from any class*
12: $\quad$**if** L[MajorityClass, 1] == 1 & sum(L[:, 1]) == 1 **then**
13: $\quad\quad$**continue** ▷ *c is excluded because it contains just a majority class outlier*
14: $\quad$**for** each class $y \in Y$ **do** ▷ *Begin class-wise cluster analysis*
15: $\quad\quad\Big(\mathbf{X}^{(c)}_y, Y^{(c)}_y\Big) \leftarrow$ contents of c belonging to y ▷ *Sub-cluster $u^{(c)}_y$ stores the samples of c that also belong to y*
16: $\quad\quad$**if** y == MajorityClass **then** ▷ *If y represents the majority class*
17: $\quad\quad\quad$**if** $L[y, 1]$ == MaxClusterSamples **then** ▷ *and it is also the majority class in c*
18: $\quad\quad\quad\quad$IncClasses $\leftarrow$ IncClasses + 1
19: $\quad\quad\quad\quad\Big(\mathbf{X}^{(c)}_{inc}, Y^{(c)}_{inc}\Big)$.extend$\Big(\mathbf{X}^{(c)}_y, Y^{(c)}_y\Big)$ ▷ *it will be included in the oversampling process (as majority class)*
20: $\quad\quad\quad$**else**
21: $\quad\quad\quad\quad\Big(\mathbf{X}', Y'\Big)$.extend$\Big(\mathbf{X}^{(c)}_y, Y^{(c)}_y\Big)$ ▷ *otherwise, it is copied to the augmented dataset*
22: $\quad\quad$**else** ▷ *otherwise, if y represents a minority class*
23: $\quad\quad\quad$**if** $L[y, 1] > 1$ **then** ▷ *and c has more than 1 samples from class y*
24: $\quad\quad\quad\quad$IncClasses $\leftarrow$ IncClasses + 1
25: $\quad\quad\quad\quad\Big(\mathbf{X}^{(c)}_{inc}, Y^{(c)}_{inc}\Big)$.extend$\Big(\mathbf{X}^{(c)}_y, Y^{(c)}_y\Big)$ ▷ *it will be included in the oversampling process (as minority class)*
26: $\quad\quad\quad$**else**
27: $\quad\quad\quad\quad\Big(\mathbf{X}', Y'\Big)$.extend$\Big(\mathbf{X}^{(c)}_y, Y^{(c)}_y\Big)$ ▷ *otherwise, it is copied to the augmented dataset*
28: $\quad$**if** IncClasses > 1 **then** ▷ *If we have more than 1 classes for oversampling, then*
29: $\quad\quad\Big(\mathbf{X}^{(c)}_{os}, Y^{(c)}_{os}\Big) \leftarrow$ oversample$\Big(\Big(\mathbf{X}^{(c)}_{inc}, Y^{(c)}_{inc}\Big)\Big)$ ▷ *perform oversampling*
30: $\quad\quad\Big(\mathbf{X}', Y'\Big)$.extend$\Big(\mathbf{X}^{(c)}_{os}, Y^{(c)}_{os}\Big)$ ▷ *append the balanced data to the augmented dataset*
31: $\quad$**else** ▷ *If we have only 1 class for oversampling, then*
32: $\quad\quad\Big(\mathbf{X}', Y'\Big)$.extend$\Big(\mathbf{X}^{(c)}_{inc}, Y^{(c)}_{inc}\Big)$ ▷ *oversampling is not possible. Copy the data to the augmented dataset*
return $(\mathbf{X}', Y')$ ▷ *Return the augmented dataset*

To specify the value of t_d and thus satisfy the third requirement, we propose the following simple heuristic: At first, we compute the pairwise distances between all samples and identify their median value. Then, we fix the distance threshold by dividing the median by 3; that is, $t_d = \tilde{d}/3$. Selecting the median instead of the mean distance limits the impact of outliers, which usually appear far away from the other samples (step 4).

Next, CBR examines each cluster $c_j \in C$ and refines the data to be fed to the oversampling mechanism (steps 7–32). Initially, c_j is organized logically into $|Y|$ sub-clusters, where $|Y|$ denotes the total number of classes. A sub-cluster $u^{(j)}_y$ of c_j contains the samples $\Big(\mathbf{X}^{(c_j)}_y, Y^{(c_j)}_y\Big)$ that belong simultaneously to the cluster c_j and the class y (step 15). An auxiliary matrix L, similar to G, stores the class distribution within c_j. The light gray rectangles of Fig. 1 depict 5 examples of how the L matrix is formed. Notice that each row of L essentially represents a sub-cluster.

When HAC terminates, several samples may remain isolated into their singleton clusters. Located too far away from the other data instances, we consider them as outliers. CBR treats the outliers in the following manner: Firstly,

the ones belonging to the majority class are discarded from the dataset (step 12). Hence, in this case, CBR performs *undersampling* of the majority class. Regarding the minority class outliers, we preserve them in the dataset to avoid reducing the population of the minority classes (step 32), but we exclude them from oversampling.

After handling the outliers, CBR attempts to balance each cluster c_j by equalizing the number of samples within each sub-cluster of c_j. For this reason, it maintains a temporary dataset $\left(\mathbf{X}_{inc}^{(c_j)}, Y_{inc}^{(c_j)}\right)$ and fills it with samples that will be used later for data generation through oversampling. Interestingly, a sub-cluster $u_y^{(j)}$ of c_j may or may not participate in the oversampling process. Participation in the oversampling process means that the samples $\left(\mathbf{X}_y^{(c_j)}, Y_y^{(c_j)}\right)$ of $u_y^{(j)}$ are copied to $\left(\mathbf{X}_{inc}^{(c_j)}, Y_{inc}^{(c_j)}\right)$ (steps 19 and 25), which will be eventually fed to the oversampling mechanism (step 29).

The sub-clusters with one or zero samples are excluded from the cluster balancing process. The idea is that if a sub-cluster $u_y^{(j)}$ has no samples at all, then c_j is considered pure and should not be distorted by generating samples from class y. On the other hand, if $u_y^{(j)}$ has just one sample, then there are no adequate samples (e.g., neighbors) to proceed to oversampling.

Moreover, CBR does not produce samples belonging to the majority class. Therefore, the sub-clusters that contain majority samples participate in the oversampling process only to provide the number of samples to be created from other classes. Notice that this is not always the case: the sub-cluster that accommodates the majority class samples does not participate in the oversampling process, unless it is also the dominant class in c_j (step 17). Otherwise, it is excluded and its contents are copied directly to the output dataset $(\mathbf{X}', Y')$ (step 21). Similarly, the contents $\left(\mathbf{X}_y^{(c)}, Y_y^{(c)}\right)$ of any sub-cluster $u_y^{(j)}$ that does not participate in the oversampling process are immediately copied to the output dataset $(\mathbf{X}', Y')$ (step 27).

Finally, notice that it may not be possible to apply oversampling to a cluster. Oversampling takes place only when the temporary dataset $\left(\mathbf{X}_{inc}^{(c)}, Y_{inc}^{(c)}\right)$ first, is not empty, and second, it contains 2 or more classes. The latter requirement is checked by using an auxiliary counter variable, called IncClasses (steps 28–32). It is this variable that prevents the outliers from being oversampled.

4. Minority Centroid Oversampling

Step 29 of Algorithm 1 refers to the oversampling mechanism that balances a cluster. Next, we present the architectural details of this mechanism.

As mentioned earlier, SMOTE is based on finding the k nearest neighbors of a sample to generate artificial data instances. Firstly, such range queries are expensive, especially when the sample vectors are of high dimensionality. Secondly, the artificial samples are randomly generated over the line that connects neighbouring points, regardless of their original distance. This may lead to datasets of limited variance and degraded quality. On the other hand, the majority of the clustering-based approaches employ k-Means for grouping the input data (e.g. k-Means SMOTE [41]). However, k-Means requires prior knowledge of the number of clusters to create, being also incapable of identifying outliers. Additionally, its performance is significantly affected by the initialization of the centroid positions.

Regarding the deep generative models, GANs [16, 29] are hard to train, suffer from the effects of mode collapse, and are quite unstable unless their hyper-parameters are tuned carefully [45]. The Variational Autoencoders try to learn the probability distribution of the input data by assuming the existence of a standard Gaussian distribution [23]. However, this is not the case with the tabular data that we are dealing with here [45].

To overcome these problems, we introduce the Minority Centroid Oversampling (MCO) approach for generating artificial data instances within imbalanced clusters. Its operation is presented in Algorithm 2.

Initially, the cluster c to be balanced is examined: its class distribution matrix L and the number of samples from the local majority class are computed (steps 2–3). L stores the number of samples of c that belong to each involved class, having dimensions $|Y^{(c)}| \times 2$. In the sequel, the sub-clusters of c are analyzed in the iterative process between steps 4 and 19. Recall that a sub-cluster $u_y^{(c)}$ contains samples that simultaneously belong to cluster c and class y. Now, if $u_y^{(c)}$ contains more than one samples and fewer than the maximum number of samples, MCO computes: i) the number of samples to be generated (step 7), and ii) its centroid point μ (step 8).

The loop of steps 11–19 generates an artificial data instance in a random point on the line that connects a random sample $\mathbf{x}_p^{(c)}$ of the current sub-cluster with its respective centroid μ. Compared to SMOTE-based methods, notice how

Algorithm 2 Minority Centroid Oversampling

Input The imbalanced dataset $(\mathbf{X}^{(c)}, Y^{(c)})$
Output The augmented dataset $(\mathbf{X}'^{(c)}, Y'^{(c)})$

1: $\left(\mathbf{X}'^{(c)}, Y'^{(c)}\right) \leftarrow [\,]$ ▹ *The dataset to be returned is initially empty*
2: $L \leftarrow \left[[y, |Y_y^{(c)}|]\right], y \in Y^{(c)}$ ▹ *In-cluster class distribution:* $|Y^{(c)}| \times 2$ *matrix with the number of samples per class*
3: MaxSamples $\leftarrow \max L[:, 1]$ ▹ *Maximum number of samples from any class*
4: **for** $y \in Y$ **do** ▹ *For each involved class*
5: **if** $1 < L[y, 1] <$ MaxSamples **then** ▹ *Can we create samples from y? y must not be the majority class in c*
6: $\left(\mathbf{X}_y^{(c)}, Y_y^{(c)}\right) \leftarrow$ contents of c belonging to y ▹ *Sub-cluster* $u_y^{(c)}$ *stores the samples of c that also belong to y*
7: SamplesToCreate $\leftarrow$ MaxSamples $- L[y, 1]$ ▹ *How many samples to create*
8: $\mu \leftarrow$ centroid $(\mathbf{X}_y^{(c)})$ ▹ *Centroid of the samples of sub-cluster* $u_y^{(c)}$
9: GenSamples $\leftarrow 0$ ▹ *Number of artificial samples*
10: $p \leftarrow 0$ ▹ *Sample selector: the sample* $\mathbf{x}_p^{(c)} \in \mathbf{X}_y^{(c)}$ *that will be used for oversampling*
11: **while** GenSamples < MaxSamples **do** ▹ *The sample generation begins*
12: $s \leftarrow$ random(0, 1)
13: $\mathbf{x}' \leftarrow \mathbf{x}_p^{(c)} + s \cdot (\mathbf{x}_p^{(c)} - \mu)$ ▹ *Generate a new artificial sample between* $\mathbf{x}_p^{(c)}$ *and* μ
14: $\mathbf{X}_y^{(c)}$.append($\mathbf{x}'$) ▹ *Append the artificial sample and its class to the augmented dataset*
15: $Y_y^{(c)}$.append(y) ▹ *Append the class to output*
16: GenSamples $\leftarrow$ GenSamples + 1 ▹ *Increase the number of artificial samples*
17: $p \leftarrow p + 1$ ▹ *Select the next sample* $\mathbf{x}_{p+1}^{(c)} \in \mathbf{X}_y^{(c)}$ *for oversampling*
18: **if** $p > L[y, 1]$ **then** ▹ *If all the samples of* $u_y^{(c)}$ *have been used for oversampling, restart*
19: $p \leftarrow 0$
return $\left(\mathbf{X}'^{(c)}, Y'^{(c)}\right)$ ▹ *Return the augmented dataset*

this approach generates more reliable samples: the samples are not generated between *any* random point and *any* random neigbor, but between a random point and the centroid μ of its respective sub-cluster.

When an artificial point is created, the next sample $\mathbf{x}_{p+1}^{(c)}$ of the sub-cluster is subsequently selected and another artificial point is created between $\mathbf{x}_{p+1}^{(c)}$ and μ. The process is repeated, until the desired number of samples has been created. Notice how the assistant variable p controls the entire process by circularly selecting all the samples in the sub-cluster. Figure 2 illustrates two representative examples, where MCO balances or ignores a cluster.

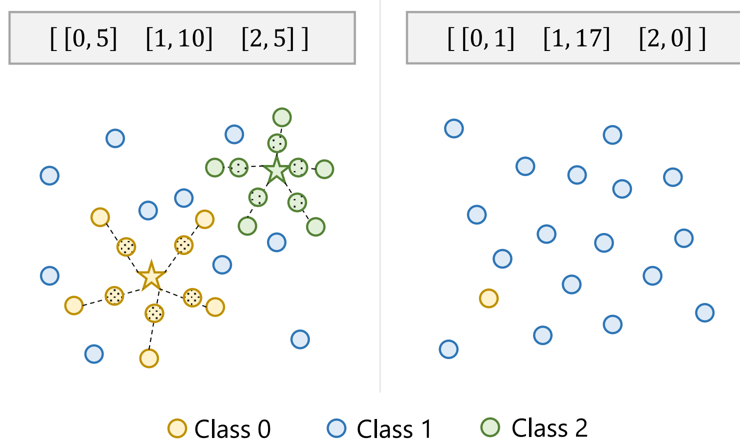

Figure 2: Two examples of MCO. In the left cluster, Class 1 is the majority class and it is not affected. For the 5 samples of Class 0, their centroid point is firstly computed. Then, 5 artificial samples are randomly created on the lines that connect each sample with their respective centroid. Similarly, 5 artificial samples are created for Class 2. In the right cluster, Class 1 is again the majority class. There is only 1 sample from Class 0 (insufficient for oversampling) and 0 samples from Class 2. The cluster is considered pure, so it is left intact.

Table 1: The characteristics of the utilized datasets.

	Dataset	Origin	n	m	IR
1	Camel-1.2	PROMISE	608	20	392:216
2	Camel-1.4	PROMISE	872	20	727:145
3	Camel-1.6	PROMISE	965	20	777:188
4	CM1	NASA MDP	498	21	449:49
5	IVY-1.1	PROMISE	111	20	48:63
6	IVY-2.0	PROMISE	352	20	312:40
7	jEdit-4.0	PROMISE	306	20	231:75
8	jEdit-4.1	PROMISE	312	20	233:79
9	jEdit-4.2	PROMISE	367	20	319:48
10	KC1	NASA MDP	2109	21	1783:326
11	KC2	NASA MDP	522	21	415:107
12	KC3	NASA MDP	458	39	415:43
13	log4j-1.0	PROMISE	135	20	101:34
14	log4j-1.1	PROMISE	109	20	72:37
15	log4j-1.2	PROMISE	205	20	16:189
16	Lucene-2.2	PROMISE	247	20	103:144
17	Lucene-2.4	PROMISE	340	20	137:203
18	PC1	NASA MDP	1109	21	1032:77
19	PC3	NASA MDP	1563	37	1403:160
20	PC4	NASA MDP	1458	37	1280:178
21	POI-1.5	PROMISE	237	20	96:141
22	POI-2.0	PROMISE	314	20	277:37
23	Velocity-1.4	PROMISE	196	20	49:147
24	Velocity-1.6	PROMISE	229	20	151:78
25	Xerces-1.2	PROMISE	440	20	369:71
26	Xerces-1.3	PROMISE	453	20	384:69
27	Xerces-1.4	PROMISE	588	20	151:437

5. Experiments

In this section we present the results of the experimental evaluation of CBR. All tests have been conducted on a commodity workstation with a CoreI7 12700K CPU (without CPU parallelization), 32GB of RAM and an NVIDIA RTX 3070 GPU. The interested reader may reproduce all the results that we present here by inspecting the implementation of CBR and the source code of all tests[1].

5.1. Datasets

Most studies on software defect detection employ two well-established collections of benchmark datasets for evaluation purposes. The first one was created by the NASA metrics data program (NASA MDP), whereas the second one is provided by the PROMISE Software Engineering Repository [39]. Both collections consist of imbalanced binary datasets that contain defective and healthy software modules. Their columns represent metric values that attempt to objectively characterize code features in terms of software quality.

Table 1 shows the 27 datasets that were used in this work. The last 3 columns denote the number of samples, the number of features and the imbalance ratio of each dataset, respectively.

5.2. Competitive Approaches

To assess the usefulness of our proposed method, we utilized 10 state-of-the-art oversampling approaches for comparison. More specifically, we considered:

[1] https://github.com/lakritidis/DeepCoreML

- 6 traditional oversampling techniques: Random Oversampling (ROS) [30], Synthetic Minority Oversampling Technique (SMOTE) [6], Borderline SMOTE [19], SVM SMOTE [32], k-Means SMOTE [9], and Adaptive Synthetic Sampling (ADASYN) [20].
- 3 state-of-the-art GANs: Conditional GAN (cGAN) [29], Safe-Borderline GAN (sbGAN) [1], and ctGAN [45]. In all cases, we used the implementations of their respective inventors. The architectures of the Discriminators and the Generators were the same for all three models. In particular, the Discriminators included two fully connected layers with 256 neurons each. Their activation functions were LeakyReLU and the logistic sigmoid, respectively. Between them, a Dropout layer was placed with weight drop probability equal to 0.5.

  The Generators comprised: i) two residual blocks, each one having a fully connected layer of output dimensionality equal to 256, ReLU activation, and an 1D batch normalization layer, and ii) another fully connected layer with tanh activation. All models were trained in batches of 32 samples for 300 epochs.
- The TabCaps classification model as it was described in [8]. TabCaps is a type of Capsule Networks, especially designed to operate on tabular data. It has achieved remarkable performance in several classification tasks. Although it is not a generative method, we decided to include it in our experiments for comparison reasons. We utilized the same architecture and set its hyper-parameters with values equal to those reported in the experimental section of [8].
- No resampling. This test was included to examine whether oversampling is actually beneficial in improving classification effectiveness.

5.3. *Classification: Model Training, Testing and Validation*

The experimental procedure was organized as follows: Initially, we considered a 3-step sequential pipeline of the form [`Resampler`, `Standardizer`, `Classifier`]. The three components were:

- `Resampler`. This step performed data augmentation for mitigating class imbalance. During each experiment, we replaced `Resampler` with either the proposed CBR method, or one of the oversampling approaches of the previous subsection.
- `Standardizer`. A typical standard scaler that performed feature normalization via the transformation $x'_i = (x_i - \mu_i)/\sigma_i$, where μ_i was the mean value of the i-th feature and σ_i its standard deviation.
- `Classifier`. The classification model that we used for defect detection. Without any loss of generality, here we employed a typical feed-forward fully connected neural network, also known as Multilayer Perceptron (MLP). The network comprised two hidden layers with 128 neurons each and ReLU activation.

  Notice that TabCaps is a classification model and not an oversampling technique. Therefore, in this case the experiments were conducted by setting `Resampler=None` and `Classifier=TabCaps`.

The examined methods were evaluated by using two measures: The first one is Balanced Accuracy, a metric that is considered ideal, especially when the underlying data is imbalanced. It is defined as the arithmetic mean of sensitivity and specificity:

$$b = (\text{Sensitivity} + \text{Specificity})/2. \tag{1}$$

In the context of software defect detection, sensitivity and specificity measure the ability of a predictor to correctly classify a module as having or not having a defect. They are defined as follows:

$$\text{Sensitivity} = \text{Recall} = TP/(TP + FN), \tag{2}$$

$$\text{Specificity} = TN/(TN + FP). \tag{3}$$

The second measure employed is Precision. It is the ratio of the correctly identified defective modules, divided by the number of all modules that were classified as defective:

$$\text{Precision} = TP/(TP + FP). \tag{4}$$

The aforementioned pipeline was evaluated in terms of Balanced Accuracy and Precision by using 5-fold cross validation in all datasets. In compliance to the established methodology, we report the average values from all 5 folds.

Table 2: Balanced Accuracy measurements. CBR is the proposed Cluster-Based Resampling method. Boldface underlined values represent the best performance, while boldface ones indicate the second best performance.

Dataset	**None**	**ROS**	**SMOTE**	**Border SMOTE**	**SVM SMOTE**	***k*-Means SMOTE**	**ADA SYN**	**CBR**	**cGAN**	**sbGAN**	**ctGAN**	**Tab* Caps**
Camel-1.2	0.594	**0.606**	0.572	0.582	0.583	0.582	0.586	**0.608**	0.583	0.594	0.582	**0.606**
Camel-1.4	0.606	0.586	0.600	0.596	0.596	0.604	0.594	**0.682**	0.598	0.593	**0.640**	0.516
Camel-1.6	0.585	0.592	0.579	0.585	0.599	0.589	0.593	**0.611**	0.567	0.584	**0.607**	0.527
CM1	0.581	0.560	0.558	0.533	0.546	0.522	0.564	**0.637**	0.557	0.587	**0.640**	0.494
IVY-1.1	0.627	**0.653**	0.617	0.601	0.638	0.603	0.622	0.645	0.609	0.632	0.633	**0.661**
IVY-2.0	0.606	0.621	0.615	0.598	**0.632**	0.582	0.604	**0.683**	0.615	0.621	0.616	0.510
jEdit-4.0	0.686	0.659	0.677	0.688	0.673	0.664	0.666	**0.702**	0.673	0.664	0.696	**0.736**
jEdit-4.1	**0.761**	0.736	0.719	0.725	0.729	0.751	0.727	0.758	0.732	0.734	**0.760**	0.674
jEdit-4.2	0.669	0.654	0.645	0.648	**0.676**	0.668	0.645	**0.704**	0.645	0.648	0.665	0.613
KC1	0.608	0.617	0.589	**0.621**	0.617	0.601	0.615	**0.629**	0.594	0.601	0.620	0.588
KC2	0.667	0.649	0.675	0.646	0.660	0.642	0.642	**0.688**	**0.681**	0.656	0.673	0.634
KC3	0.641	0.667	0.667	**0.688**	0.645	0.610	0.654	**0.688**	0.662	0.645	0.658	0.530
log4j-1.0	0.660	0.671	**0.688**	0.679	**0.690**	0.667	0.638	0.685	0.669	0.684	0.659	0.671
log4j-1.1	0.660	0.667	0.681	0.675	0.680	**0.686**	0.663	0.674	0.681	0.680	**0.698**	0.679
log4j-1.2	0.617	**0.632**	0.621	0.627	**0.629**	0.624	0.621	0.620	0.584	0.617	0.605	0.500
Lucene-2.2	0.576	0.590	0.597	0.593	0.602	0.584	0.596	0.588	0.609	**0.610**	0.579	**0.617**
Lucene-2.4	0.671	**0.672**	0.640	0.655	0.658	0.670	0.641	**0.674**	0.663	0.645	0.647	0.669
PC1	0.639	0.624	0.593	0.627	**0.658**	0.649	0.587	**0.677**	0.536	0.554	0.578	0.603
PC3	0.645	0.662	0.655	0.643	0.647	**0.673**	0.647	0.653	0.651	0.646	**0.674**	0.577
PC4	0.770	**0.785**	0.776	0.759	**0.790**	0.780	0.772	0.765	0.779	0.749	**0.785**	0.741
POI-1.5	0.709	0.696	0.693	0.709	**0.717**	0.689	0.705	**0.727**	0.716	0.691	0.686	0.708
POI-2.0	0.592	0.603	0.562	0.583	0.585	0.607	0.603	0.610	0.589	0.589	**0.640**	**0.701**
Velocity-1.4	0.760	0.754	0.751	0.758	0.765	0.748	0.748	**0.774**	0.767	**0.787**	0.767	0.714
Velocity-1.6	0.696	0.663	0.676	0.686	0.666	0.659	0.669	**0.717**	0.683	0.680	**0.698**	0.575
Xerces-1.2	0.634	**0.715**	0.684	0.698	0.696	**0.706**	0.697	0.634	0.626	0.587	0.648	0.544
Xerces-1.3	0.653	0.686	0.684	0.695	0.667	0.657	0.679	**0.775**	0.675	0.689	0.690	**0.714**
Xerces-1.4	0.860	0.856	0.856	0.853	**0.861**	0.857	0.855	0.856	0.852	0.846	**0.868**	0.860

5.4. Balanced Accuracy Results

Table 2 presents the Balanced Accuracy measurements of our MLP classifier, when it is combined with the examined methods in all 27 datasets. The second column reports the Balanced Accuracy of the MLP classifier without applying any data augmentation technique (that is, when `Resampler=None` in the pipeline). Furthermore, the box plot of Figure 3 illustrates a performance comparison of the involved techniques.

When no data augmentation was applied, the measured values were, in some cases, unexpectedly high. For example, in Xerces-1.4 the Balanced Accuracy was equal to 0.86, even though the imbalance ratio of this dataset is about 1:3. This is an indication of the classification effectiveness of neural networks in the presence of imbalanced datasets. Therefore, by selecting MLP as test classifier, it becomes more difficult for a data augmentation method to exhibit its usefulness.

CBR outperformed all the 11 adversary methods in 13 out of 27 datasets, that is, in approximately half of our experimental universe. Moreover, it achieved the second best performance in 3 other cases. The box plot of Figure 3 reveals that the median Balanced Accuracy value of CBR (orange line) was higher even than the median Balanced Accuracy value of the upper quartile (top border of the boxes) of all its adversary methods.

The strongest opponent was ctGAN, a model that is considered in the current literature as one of the most effective in tabular data generation. ctGAN was the best method in 4 out of 27 datasets. It was also ranked second in 6 other cases. These results are indicative of the high effectiveness of CBR, since it outperformed the state-of-the-art ctGAN in terms of Balanced Accuracy by a large margin.

On the other hand, ctGAN was the slowest among all the examined methods. Even with GPU acceleration, it was many times slower than CBR and the 6 traditional oversampling techniques. Also notice that ctGAN generates additional columns because of the way it encodes the continuous variables. This increase in input data dimensionality is not infinitesimal and renders the model slower than cGAN and sbGAN too. Detailed measurements of the time efficiency of the examined methods are presented in Subsection 5.7.

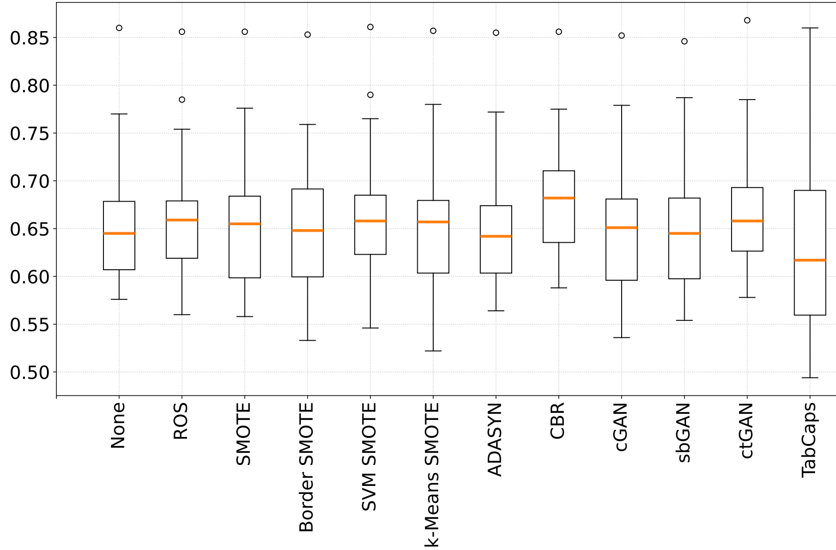

Figure 3: Box plot of the Balanced Accuracy achieved by the examined data augmentation techniques.

TabCaps exhibited mixed performance. On one hand, it achieved top performance in four datasets and the second highest Balanced Accuracy in other two. On the other, it was particularly ineffective in multiple datasets, e.g., Camel-1.4, CM1, log4j-1.2 and others. In total, TabCaps scored the lowest Balanced Accuracy than all the other methods in 12 cases. This highlights its sensitivity on the input dataset and perhaps, the tuning of its hyper-parameters.

Interestingly, the only competitive method that performs clustering, k-Means SMOTE never achieved the best performance; it just occupied the second place in 3 cases. This is another indication that simple clustering alone cannot yield top results. However, when cluster structure analysis is performed, accompanied by outlier handling and several heuristics, then clustering may indeed be rendered robust.

Regarding the other methods, Random Oversampling was the best method in 2 datasets and the second best in 4 other cases. SVM-SMOTE exhibited a similar effectiveness, as it also occupied the first and second position in 2 and 4 tests respectively. As for Generative Adversarial Nets, sbGAN achieved the highest performance in one case, whereas the Conditional GAN was ranked second only in one dataset.

ADASYN and SMOTE were among the weakest methods in this experiment. The former was never among the best two performing methods, whereas the latter was ranked second only in the log4j-1.0 dataset. Moreover, both of them had beneficiary effects on classification performance only in 11 out of 27 datasets (i.e., their Balanced Accuracy was higher than that of the "None" case). This verifies the findings of other researchers who concluded that oversampling is not always beneficiary for classification performance, especially when an imbalance-resistant classifier (such as our MLP) is utilized.

5.5. Precision Results

We proceed with the presentation of the Precision measurements. Table 3 contains the performances of our examined methods on the 27 benchmark datasets, whereas Figure 4 offers a comparative illustration.

Similarly to the previous experiments, CBR achieved the highest performance compared to all the traditional oversampling techniques, the three generative models, and TabCaps. More specifically, its Precision was the highest in 9 out of 27 datasets, and the second best in 5 other cases. In general, our proposed method exhibited good behaviour in all experiments. This is graphically demonstrated by the box plots of Figure 4.

The two strongest opponents of CBR in terms of Precision were cGAN and SVM SMOTE. Both methods achieved decent performance in all experiments, albeit lower than that of CBR. In particular, the former outperformed all the other approaches in 5 datasets, whereas it was ranked second in 3 other cases. The latter scored the highest Precision in two datasets and the second highest in three tests.

Table 3: Precision measurements. CBR is the proposed Cluster-Based Resampling method. Boldface underlined values represent the best performance, while boldface ones indicate the second best performance.

Dataset	None	ROS	SMOTE	Border SMOTE	SVM SMOTE	*k*-Means SMOTE	ADA SYN	CBR	cGAN	sbGAN	ctGAN	Tab* Caps
Camel-1.2	0.630	**0.640**	0.609	0.618	0.619	0.619	0.623	**0.640**	0.623	0.633	0.618	0.594
Camel-1.4	0.792	0.776	0.779	0.777	0.779	0.785	0.776	**0.821**	0.788	0.785	**0.799**	0.285
Camel-1.6	0.746	0.746	0.738	0.742	0.748	0.745	0.747	**0.752**	0.735	0.744	**0.752**	0.555
CM1	0.860	0.843	0.843	0.835	0.843	0.830	0.847	**0.881**	0.838	0.867	**0.871**	0.311
IVY-1.1	0.636	**0.667**	0.626	0.612	0.646	0.612	0.633	0.655	0.617	0.641	0.642	**0.877**
IVY-2.0	0.858	0.855	0.851	0.846	**0.859**	0.838	0.847	**0.869**	0.856	0.850	0.841	0.198
jEdit-4.0	**0.791**	0.766	0.776	0.785	0.763	0.767	0.767	0.781	0.776	0.773	0.774	**0.788**
jEdit-4.1	**0.829**	0.804	0.789	0.794	0.797	**0.816**	0.799	0.814	0.804	0.811	0.811	0.686
jEdit-4.2	0.859	0.859	0.854	0.861	**0.867**	0.857	0.853	**0.865**	0.850	0.853	0.843	0.566
KC1	0.800	0.797	0.783	0.797	0.796	0.788	0.794	0.801	**0.807**	**0.807**	0.804	0.700
KC2	0.791	0.770	0.785	0.765	0.776	0.765	0.764	**0.794**	**0.805**	0.785	0.788	0.000
KC3	0.887	0.889	0.889	**0.892**	0.882	0.871	0.884	**0.894**	**0.892**	0.890	0.890	0.144
log4j-1.0	0.771	0.773	0.777	0.779	**0.789**	0.776	0.750	**0.787**	0.778	0.783	0.758	0.667
log4j-1.1	0.696	0.702	0.710	0.712	0.715	**0.727**	0.700	0.705	0.713	0.718	**0.721**	0.717
log4j-1.2	0.900	**0.903**	0.897	0.899	0.902	0.897	0.897	0.900	0.888	0.900	0.880	**0.924**
Lucene-2.2	0.593	0.608	0.614	0.610	0.621	0.607	0.616	0.606	**0.628**	0.627	0.595	**0.794**
Lucene-2.4	0.693	0.693	0.661	0.677	0.678	0.688	0.661	**0.696**	0.685	0.666	0.669	**0.798**
PC1	**0.913**	0.904	0.899	0.906	0.912	0.909	0.898	**0.916**	0.881	0.886	0.894	0.233
PC3	0.875	0.874	0.872	0.867	0.868	0.876	0.867	0.868	**0.878**	0.875	**0.877**	0.293
PC4	0.908	0.907	0.909	0.902	**0.911**	0.909	0.905	0.901	**0.912**	0.901	0.904	0.400
POI-1.5	0.723	0.713	0.705	0.721	**0.729**	0.702	0.718	**0.740**	**0.729**	0.715	0.700	0.644
POI-2.0	**0.855**	0.837	0.818	0.834	0.847	0.837	0.837	0.842	0.836	0.839	**0.851**	0.600
Velocity-1.4	0.848	0.846	0.820	0.828	0.835	0.827	0.821	0.853	0.838	**0.862**	0.819	**0.893**
Velocity-1.6	**0.734**	0.700	0.713	0.724	0.702	0.698	0.708	**0.743**	0.727	0.724	0.733	0.496
Xerces-1.2	0.813	**0.837**	0.821	0.827	0.831	**0.832**	0.829	0.799	0.808	0.785	0.812	0.167
Xerces-1.3	0.831	0.837	0.837	0.839	0.827	0.825	0.832	**0.876**	0.842	0.848	0.834	**0.971**
Xerces-1.4	0.892	0.889	0.889	0.887	0.889	0.890	0.882	0.888	0.887	0.884	**0.894**	**0.929**

In contrast, ctGAN, the second best model in terms of Balanced Accuracy, did not yield top results in this experiment. Hence, it was the best method in only one dataset (Camel-1.6) and the second best in 6 other cases. Among the other methods, ctGAN was on average ranked 6th in terms of achieved Precision values. Regarding our third attested generative model, sbGAN dominated over all its adversary approaches in one dataset (KC1) and it was ranked second

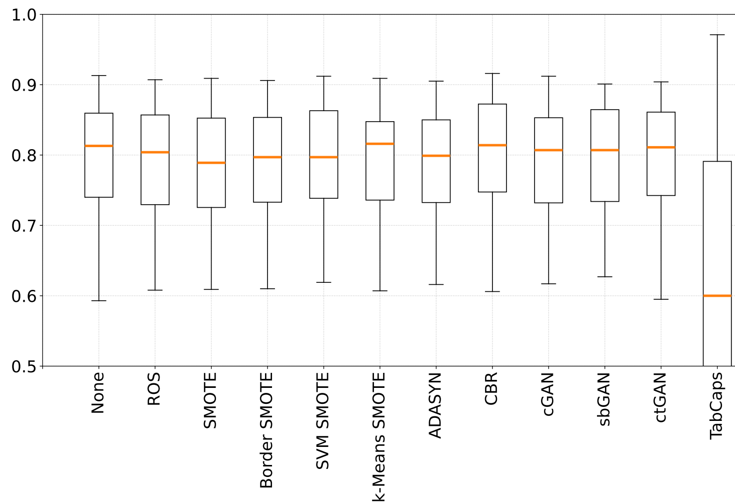

Figure 4: Box plot of the Precision values achieved by the examined data augmentation techniques.

Table 4: Mean rankings (lower is better) of the examined methods in terms of Balanced Accuracy (column 2) and Precision (column 3).

Method	Mean Rank (Balanced Accuracy)	Mean Rank (Precision)
CBR	3.185185 (1)	3.777778 (1)
ctGAN	5.000000 (2)	6.407407 (6)
SVM SMOTE	5.277778 (3)	5.666667 (2)
ROS	5.685185 (4)	6.203704 (5)
Border SMOTE	7.055556 (5)	7.333333 (7)
***k*-Means SMOTE**	7.185185 (6)	7.333333 (7)
cGAN	7.222222 (7)	5.833333 (3)
SMOTE	7.296296 (8)	7.833333 (9)
sbGAN	7.351852 (9)	5.870370 (4)
ADASYN	7.907407 (10)	8.333333 (10)
TabCaps	8.277778 (11)	8.481481 (11)

in Velocity-1.4.

Similarly to the previous experiment, the performance of TabCaps had significant divergences from dataset to dataset. More specifically, the model achieved excellent measurements of Precision in 7 datasets. However, in 10 other cases its Precision values were (much) below 0.5. Unfortunately, as happened in the experiments of Balanced Accuracy, TabCaps was on average the worst method among all. However, its top performance in several tests highlights its usefulness, while the bad results may be indicative of bad hyper-parameter tuning.

Another significant observation is that in three cases, the highest Precision was obtained when no oversampling technique was applied. "None" was also the second best strategy in terms of Precision on two other datasets. Similarly to the previous experiment, ADASYN and SMOTE were again among the two weakest methods.

Finally, Table 4 conveniently summarizes the performances of the 11 data augmentation approaches in terms of Balanced Accuracy (column 2), and Precision (column 3). The numbers inside the parentheses denote the ranking of the methods in this comparison. In the first case, it is shown that CBR was the method with the highest Balanced Accuracy; its average ranking in all 27 measurements was approximately 3.19. In comparison, the average ranking of the second best method, ctGAN, was exactly 5.0, followed by SVM SMOTE with an average ranking of 5.28.

Similar comments can be made for the third column that represents the Precision measurements. CBR was again the highest performing method with an average ranking of 3.78. Nevertheless, there are changes in the ranking of the algorithms below the first position. Therefore, SVM SMOTE was the method that achieved the second best Precision measurements; on average, it was ranked 5.67-th.

The weakness of TabCaps, SMOTE and ADASYN in both Balanced Accuracy and Precision experiments is demonstrated here. These methods consistently occupied the three last positions in the respective rankings.

5.6. Statistical Significance Tests

To examine the significance of our results, we performed two statistical analysis tests on both the Balanced Accuracy and Precision measurements. For this purpose, we employed STAC[2], an online statistical tool for algorithm comparison [37]. In the following analysis, the significance level was considered equal to 0.05.

At first, the truth of the null hypothesis H_0 *"There is no statistical difference in the performance results of all algorithms"* was attested by executing the Friedman test. The test returned p-values equal to $1.658 \cdot 10^{-6}$ for Balanced Accuracy and $3.736 \cdot 10^{-6}$ for Precision, indicating the rejection of H_0.

We also conducted a post-hoc, non-parametric pairwise analysis by executing the Finner test on the obtained measurements of Balanced Accuracy and Precision [12]. According to the experimental study of García et al., the Finner test yields better results compared to other statistical tests (like Bonferroni and Holm), while it is easy to comprehend [13].

[2]https://tec.citius.usc.es/stac/

Table 5: p-values of the Finner pairwise post-hoc test on the Balanced Accuracy results. Bold fonts denote statistically significant measurements.

Method	None	ROS	SMOTE	Border SMOTE	SVM SMOTE	k-Means SMOTE	ADA SYN	CBR	cGAN	sbGAN	ctGAN	Tab Caps
None	–	0.5223	0.5843	0.6908	0.3249	0.6216	0.2936	**0.0049**	0.6062	0.5550	0.2252	0.1769
ROS	0.5223	–	0.2081	0.2914	0.7497	0.2368	0.0883	**0.0491**	0.2269	0.1920	0.6062	**0.0412**
SMOTE	0.5843	0.2081	–	0.8455	0.1097	0.9196	0.6272	**0.0005**	0.9449	0.9570	**0.0472**	0.4672
Border SMOTE	0.6908	0.2914	0.8455	–	0.1626	0.9126	0.5262	**0.0008**	0.8937	0.8165	0.1047	0.3476
SVM SMOTE	0.3249	0.7497	0.1097	0.1626	–	0.1313	**0.0399**	0.1047	0.1254	0.1047	0.8242	**0.0147**
k-Means SMOTE	0.6216	0.2368	0.9196	0.9126	0.1313	–	0.5888	**0.0005**	0.9699	0.8937	0.0883	0.4149
ADASYN	0.2936	0.0883	0.6272	0.5262	**0.0399**	0.5888	–	**0.0001**	0.6062	0.6587	**0.0182**	0.7697
CBR	**0.0049**	0.0501	**0.0005**	**0.0008**	0.1047	**0.0005**	**0.0001**	–	**0.0005**	**0.0005**	0.1555	$< 10^{-4}$
cGAN	0.6062	0.2269	0.9449	0.8937	0.1254	0.9699	0.6062	**0.0005**	–	0.9126	0.0883	0.4293
sbGAN	0.5550	0.1920	0.9570	0.8165	0.1047	0.8937	0.6587	**0.0005**	0.9126	–	0.0708	0.4944
ctGAN	0.2252	0.6062	0.0772	0.1047	0.8242	0.0883	**0.0182**	0.1555	0.0883	0.0708	–	**0.0061**
TabCaps	0.1769	**0.0412**	0.4672	0.3476	**0.0147**	0.4149	0.7697	$< 10^{-4}$	0.4293	0.4944	**0.0061**	–

Table 6: p-values of the Finner pairwise post-hoc test on the Precision results. Bold fonts denote statistically significant measurements.

Method	None	ROS	SMOTE	Border SMOTE	SVM SMOTE	k-Means SMOTE	ADA SYN	CBR	cGAN	sbGAN	ctGAN	Tab Caps
None	–	0.3108	**0.0249**	**0.0443**	0.5461	**0.0443**	**0.0049**	0.3743	0.4484	0.4441	0.2432	**0.0048**
ROS	0.3108	–	0.1948	0.3743	0.6579	0.3743	0.0913	**0.0443**	0.7456	0.7671	0.8582	0.0687
SMOTE	**0.0249**	0.1948	–	0.6707	0.0871	0.6707	0.6707	**0.0008**	0.1060	0.1114	0.2457	0.5945
Border SMOTE	**0.0443**	0.3743	0.6707	–	0.1920	1.0000	0.4246	**0.0048**	0.2432	0.2432	0.4484	0.3743
SVM SMOTE	0.5461	0.6579	0.0871	0.1920	–	0.1920	**0.0426**	0.1232	0.8774	0.8582	0.5461	**0.0299**
k-Means SMOTE	**0.0443**	0.3743	0.6707	1.0000	0.1920	–	0.4246	**0.0048**	0.2432	0.2432	0.4484	0.3743
ADASYN	**0.0049**	0.0913	0.6707	0.4246	**0.0426**	0.4246	–	**0.0001**	**0.0421**	**0.0481**	0.1171	0.8877
CBR	0.3743	0.0543	**0.0008**	**0.0048**	0.1232	**0.0048**	**0.0001**	–	0.0964	0.0957	**0.0426**	**0.0001**
cGAN	0.4484	0.7456	0.1060	0.2432	0.8774	0.2432	**0.0421**	0.0964	–	0.9715	0.6388	**0.0426**
sbGAN	0.4441	0.7671	0.1114	0.2432	0.8582	0.2432	**0.0481**	0.0957	0.9715	–	0.6579	**0.0426**
ctGAN	0.2432	0.8582	0.2457	0.4484	0.5461	0.4484	0.1171	**0.0426**	0.6388	0.6579	–	0.0960
TabCaps	**0.0048**	0.0687	0.5945	0.3743	**0.0299**	0.3743	0.8877	**0.0001**	**0.0426**	**0.0426**	0.0960	–

The results of the Finner tests are reported in Tables 5 and 6 for Balanced Accuracy and Precision, respectively. The readings in bold fonts indicate statistically significant measurements. The obtained p-values reveal that CBR performs differently compared to all the other resampling methods. In combination with the results of the previous subsections, we may safely conclude that the superiority of CBR in the aforementioned experiments was not random.

In contrast, the results of the Finner tests regarding the adversary resampling methods indicate performance differences that, in most cases, were statistically insignificant.

5.7. Execution Times

We also performed an experimental study of the time efficiency of the involved methods. Table 7 presents the execution times of CBR compared to the ones of the other oversampling approaches. Since our experiments have been conducted by using 5-fold cross validation, these times concern the mean duration of oversampling for all 5 folds.

The results reveal that CBR is at least as fast as SMOTE, its variants, and ADASYN. In fact, it was proved that CBR outperformed these methods in all datasets, even by small margins. Despite our approach applies agglomerative clustering on the underlying data (cubic time complexity), it is still capable of achieving better running times, because it avoids the expensive nearest neighbor searches used by the other methods.

Table 7: Method execution times.

Dataset	ROS	SMOTE	Border SMOTE	SVM SMOTE	*k*-Means SMOTE	ADASYN	CBR	cGAN	sbGAN	ctGAN
Camel-1.2	0.69	0.62	0.58	0.67	0.63	0.64	0.61	8.61	7.74	41.77
Camel-1.4	1.09	0.99	0.98	0.98	1.10	1.07	0.96	13.57	7.88	59.00
Camel-1.6	1.08	0.95	0.84	0.98	1.23	0.85	0.90	13.57	10.13	67.00
CM1	0.77	0.68	0.63	0.61	0.70	0.66	0.53	7.26	3.35	35.66
IVY-1.1	0.15	0.16	0.15	0.15	0.16	0.15	0.14	1.68	1.70	5.85
IVY-2.0	0.48	0.48	0.43	0.43	0.51	0.48	0.32	5.01	2.54	22.68
jEdit-4.0	0.39	0.40	0.38	0.41	0.39	0.39	0.33	4.42	3.19	19.80
jEdit-4.1	0.39	0.39	0.36	0.37	0.38	0.36	0.34	4.40	3.17	19.85
jEdit-4.2	0.37	0.38	0.35	0.44	0.34	0.35	0.28	5.41	2.75	25.25
KC1	2.56	2.31	2.05	2.27	1.90	2.43	1.93	31.32	15.35	154.97
KC2	0.70	0.63	0.69	0.71	0.67	0.63	0.56	7.97	4.33	38.66
KC3	0.52	0.48	0.53	0.51	0.53	0.50	0.37	6.82	3.12	48.91
log4j-1.0	0.14	0.13	0.12	0.14	0.14	0.13	0.11	2.16	1.66	8.60
log4j-1.1	0.15	0.15	0.14	0.15	0.16	0.15	0.14	1.69	1.57	5.79
log4j-1.2	0.18	0.18	0.19	0.22	0.19	0.19	0.16	3.32	1.71	14.29
Lucene-2.2	0.31	0.30	0.32	0.32	0.32	0.30	0.28	3.82	3.33	16.97
Lucene-2.4	0.37	0.39	0.39	0.39	0.38	0.39	0.36	4.88	4.43	22.42
PC1	1.22	1.27	1.14	1.03	1.08	1.19	0.95	17.68	5.23	77.01
PC3	1.09	1.21	1.07	1.13	1.11	1.18	0.95	21.33	8.34	157.40
PC4	1.38	1.22	1.21	1.19	1.10	1.20	1.12	19.87	8.96	145.73
POI-1.5	0.24	0.26	0.27	0.29	0.29	0.25	0.19	3.30	3.17	14.22
POI-2.0	0.45	0.43	0.39	0.41	0.38	0.41	0.31	4.45	2.41	19.83
Velocity-1.4	0.27	0.30	0.27	0.29	0.28	0.27	0.16	2.78	1.69	11.44
Velocity-1.6	0.30	0.31	0.31	0.32	0.31	0.32	0.28	3.34	2.92	14.23
Xerces-1.2	0.51	0.46	0.47	0.39	0.48	0.37	0.40	5.92	3.85	30.78
Xerces-1.3	0.50	0.47	0.47	0.54	0.52	0.47	0.38	6.65	3.23	30.83
Xerces-1.4	0.70	0.70	0.66	0.69	0.71	0.72	0.55	8.12	4.30	39.06

Moreover, as anticipated, CBR is much faster than all Generative Adversarial Nets, and especially ctGAN. In particular, CBR outperforms the conditional GAN and SB-GAN in terms of execution times by more than an order of magnitude. On the other hand, ctGAN was the slowest model: the way this model encodes the numerical features (by using a Bayesian Gaussian Mixture) increases the dataset dimensionality, leading to significantly degraded execution durations.

6. Conclusions and Future Work

In this paper we introduced a new clustering-based data resampling technique called CBR. The proposed method was designed to effectively handle imbalanced data in software defect detection tasks. However, it is applicable to all multi-class imbalanced classification problems. One of the most powerful features of CBR is that it can simultaneously perform undersampling of the majority class and oversampling of the minority classes.

Initially, a distance threshold is automatically specified by applying a simple heuristic that first, computes the distances between all pairs of samples and then, divides their median by 3. Next, agglomerative clustering is applied to the imbalanced dataset by preventing clusters to be merged in case their distance is larger than the aforementioned threshold. At the time clustering is completed, the remaining singleton clusters are treated as outliers (because they are too far away to be merged with other samples). The majority class outliers are evicted from the dataset (thus performing undersampling), whereas the minority class outliers are preserved, but are not used as reference for oversampling.

In the sequel, the algorithm attempts to balance each cluster by applying conditional oversampling. The conditions dictate how oversampling will or will not take place. For example, if the majority of the points in a cluster belong to a minority class, then these points are also not used for oversampling. Furthermore, instead of creating artificial data instances between a sample and its nearest neighbors, as SMOTE and k-Means SMOTE do, we synthesize data between a sample and its respective centroid. In this manner, we avoid the costly spatial k-NN queries, creating also data of better quality; e.g., we avoid generating samples between two distant neighbors.

CBR has been extensively evaluated by using 27 well-established datasets from the research area of software engineering. The proposed method has demonstrated superior Balanced Accuracy, Precision and running times compared to a wide range of adversary approaches. In particular, we compared it against six traditional oversampling approaches (Random Oversampling, SMOTE, Borderline SMOTE, SVM SMOTE, k-Means SMOTE, and ADASYN), three state-of-the-art Generative Adversarial Networks (Conditional GAN, Safe-Borderline GAN, and ctGAN) and TabCaps, a Capsule Network designed for classifying tabular data.

In the context of software defect detection, the aforementioned heuristic specifies the cluster distance threshold in a satisfactory manner. However, a more extensive case study is required to examine its performance in other applications. Moreover, CBR applies the traditional agglomerative clustering algorithm to effectively identify clusters of similar data instances. Despite this is a general case method with acceptable performance, it does not provide information on the underlying probability distribution of the input data. Other clustering algorithms, like the Gaussian Mixture Model, could provide an attractive alternative.

Our future work includes experiments that will combine the strong features of GANs with the cluster structure analysis strategy of CBR. More specifically, ctGAN is ingeniously modelling the continuous variables with a Bayesian mixture of Gaussians. However, the additional columns created render the model slow. Moreover, ctGAN does not take into consideration the outliers. These elements provide an additional motivation for further research on the area.

References

[1] L. Akritidis, A. Fevgas, M. Alamaniotis, P. Bozanis, Conditional Data Synthesis with Deep Generative Models for Imbalanced Dataset Oversampling, in: Proceedings of the 35th IEEE International Conference on Tools with Artificial Intelligence, 444–451, 2023.

[2] Ö. F. Arar, K. Ayan, Software defect prediction using Cost-Sensitive Neural Network, Applied Soft Computing 33 (2015) 263–277.

[3] V. R. Basili, L. C. Briand, W. L. Melo, A validation of object-oriented design metrics as quality indicators, IEEE Transactions on Software Engineering 22 (10) (1996) 751–761.

[4] C. Bunkhumpornpat, K. Sinapiromsaran, C. Lursinsap, Safe-Level-SMOTE: Safe-level-synthetic minority over-sampling technique for handling the class imbalanced problem, in: Proceedings of the 13th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, 475–482, 2009.

[5] N. V. Chawla, Data mining for imbalanced datasets: An overview, Data Mining and Knowledge Discovery Handbook (2010) 875–886.

[6] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, SMOTE: Synthetic Minority Over-sampling Technique, Journal of Artificial Intelligence Research 16 (2002) 321–357.

[7] N. V. Chawla, A. Lazarevic, L. O. Hall, K. W. Bowyer, SMOTEBoost: Improving prediction of the minority class in boosting, in: Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases, 107–119, 2003.

[8] J. Chen, K. Liao, Y. Fang, D. Chen, J. Wu, TabCaps: A Capsule Neural Network for Tabular Data Classification with BoW Routing, in: Proceedings of the 11th International Conference on Learning Representations, 1–15, 2023.

[9] G. Douzas, F. Bacao, F. Last, Improving imbalanced learning through a heuristic oversampling method based on k-means and SMOTE, Information Sciences 465 (2018) 1–20.

[10] K. O. Elish, M. O. Elish, Predicting defect-prone software modules using Support Vector Machines, Journal of Systems and Software 81 (5) (2008) 649–660.

[11] A. Fernández, S. Garcia, F. Herrera, N. V. Chawla, SMOTE for learning from imbalanced data: progress and challenges, marking the 15-year anniversary, Journal of Artificial Intelligence Research 61 (2018) 863–905.

[12] H. Finner, On a monotonicity problem in step-down multiple test procedures, Journal of the American Statistical Association 88 (423) (1993) 920–923.

[13] S. García, A. Fernández, J. Luengo, F. Herrera, Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power, Information Sciences 180 (10) (2010) 2044–2064.

[14] S. García, J. Luengo, F. Herrera, Tutorial on practical tips of the most influential data preprocessing algorithms in data mining, Knowledge-Based Systems 98 (2016) 1–29.

[15] N. Gayatri, S. Nickolas, A. Reddy, S. Reddy, A. Nickolas, Feature selection using Decision Tree induction in class level metrics dataset for software defect predictions, in: Proceedings of the World Congress on Engineering and Computer Science, vol. 1, 124–129, 2010.

[16] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, Generative Adversarial Nets, Advances in Neural Information Processing Systems 27 (2014) 2672–2680.

[17] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, Generative Adversarial Networks, Communications of the ACM 63 (11) (2020) 139–144.

[18] J. Goyal, R. Ranjan Sinha, Software defect-based prediction using Logistic Regression: Review and challenges, in: Proceedings of the 2nd International Conference on Sustainable Technologies for Computational Intelligence, 233–248, 2022.
[19] H. Han, W.-Y. Wang, B.-H. Mao, Borderline-SMOTE: A new over-sampling method in imbalanced data sets learning, in: Proceedings of 2005 International Conference on Intelligent Computing (Advances in Intelligent Computing), 878–887, 2005.
[20] H. He, Y. Bai, E. A. Garcia, S. Li, ADASYN: Adaptive Synthetic Sampling approach for imbalanced learning, in: Proceedings of the 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence), 1322–1328, 2008.
[21] R. Jindal, R. Malhotra, A. Jain, Software defect prediction using Neural Networks, in: Proceedings of the 3rd International Conference on Reliability, Infocom Technologies and Optimization, 1–6, 2014.
[22] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, J. Liu, Dictionary learning based software defect prediction, in: Proceedings of the 36th International Conference on Software Engineering, 414–423, 2014.
[23] D. P. Kingma, M. Welling, Auto-encoding Variational Bayes, arXiv preprint arXiv:1312.6114 .
[24] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, IEEE Transactions on Software Engineering 34 (4) (2008) 485–496.
[25] J. Li, P. He, J. Zhu, M. R. Lyu, Software defect prediction via Convolutional Neural Network, in: Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security, 318–328, 2017.
[26] H. Liang, Y. Yu, L. Jiang, Z. Xie, Seml: A semantic LSTM model for software defect prediction, IEEE Access 7 (2019) 83812–83824.
[27] W.-C. Lin, C.-F. Tsai, Y.-H. Hu, J.-S. Jhang, Clustering-based undersampling in class-imbalanced data, Information Sciences 409 (2017) 17–26.
[28] C. Manjula, L. Florence, Deep neural network based hybrid approach for software defect prediction using software metrics, Cluster Computing 22 (Suppl 4) (2019) 9847–9863.
[29] M. Mirza, S. Osindero, Conditional Generative Adversarial Nets, arXiv preprint arXiv:1411.1784 .
[30] A. Moreo, A. Esuli, F. Sebastiani, Distributional random oversampling for imbalanced text classification, in: Proceedings of the 39th ACM SIGIR International Conference on Research and Development in Information Retrieval, 805–808, 2016.
[31] H. S. Munir, S. Ren, M. Mustafa, C. N. Siddique, S. Qayyum, Attention based GRU-LSTM for software defect prediction, Plos one 16 (3) (2021) e0247444.
[32] H. M. Nguyen, E. W. Cooper, K. Kamei, Borderline over-sampling for imbalanced data classification, International Journal of Knowledge Engineering and Soft Data Paradigms 3 (1) (2011) 4–21.
[33] S. Omri, C. Sinz, Deep learning for software defect prediction: A survey, in: Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering, 209–214, 2020.
[34] Y. Peng, G. Wang, H. Wang, User preferences based software defect detection algorithms selection using MCDM, Information Sciences 191 (2012) 3–13.
[35] L. Perreault, S. Berardinelli, C. Izurieta, J. Sheppard, Using classifiers for software defect detection, in: Proceedings of the ISCA 26th International Conference on Software Engineering and Data Engineering, 2–4, 2017.
[36] L. Qiao, X. Li, Q. Umer, P. Guo, Deep learning based software defect prediction, Neurocomputing 385 (2020) 100–110.
[37] I. Rodríguez-Fdez, A. Canosa, M. Mucientes, A. Bugarín, STAC: a web platform for the comparison of algorithms using statistical tests, in: Proceedings of the 2015 IEEE International Conference on Fuzzy Systems, 1–8, 2015.
[38] S. Sabour, N. Frosst, G. E. Hinton, Dynamic routing between Capsules, Advances in Neural Information Processing Systems 30.
[39] J. Sayyad Shirabad, T. Menzies, The PROMISE Repository of Software Engineering Databases, School of Information Technology and Engineering, University of Ottawa, Canada", URL `http://promise.site.uottawa.ca/SERepository`, 2005.
[40] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, A. Napolitano, RUSBoost: A hybrid approach to alleviating class imbalance, IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans 40 (1) (2009) 185–197.
[41] J. Song, X. Huang, S. Qin, Q. Song, A bi-directional sampling based on K-means method for imbalance text classification, in: Proceedings of the 15th IEEE/ACIS International Conference on Computer and Information Science, 1–5, 2016.
[42] V. Vashisht, M. Lal, G. Sureshchandar, et al., A framework for software defect prediction using Neural Networks, Journal of Software Engineering and Applications 8 (8) (2015) 384–394.
[43] J. Wang, B. Shen, Y. Chen, Compressed C4.5 models for software defect prediction, in: Proceedings of the 12th International Conference on Quality Software, 13–16, 2012.
[44] S. Wang, X. Yao, Diversity analysis on imbalanced data sets by using ensemble models, in: Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Data Mining, 324–331, 2009.
[45] L. Xu, M. Skoularidou, A. Cuesta-Infante, K. Veeramachaneni, Modeling tabular data using Conditional GAN, Advances in Neural Information Processing Systems 32.
[46] S.-J. Yen, Y.-S. Lee, Cluster-based under-sampling approaches for imbalanced data distributions, Expert Systems with Applications 36 (3) (2009) 5718–5727.
[47] H. Zhang, L. Huang, C. Q. Wu, Z. Li, An effective Convolutional Neural Network based on SMOTE and Gaussian mixture model for intrusion detection in imbalanced dataset, Computer Networks 177 (2020) 107315.
[48] L. Zhao, Z. Shang, L. Zhao, T. Zhang, Y. Y. Tang, Software defect prediction via cost-sensitive Siamese parallel fully-connected neural networks, Neurocomputing 352 (2019) 64–74.