Information and Software Technology 129 (2021) 106432

Contents lists available at ScienceDirect

Information and Software Technology

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

COSTE: Complexity-based OverSampling TEchnique to alleviate the class imbalance problem in software defect prediction

Shuo Feng [a,*], Jacky Keung [a,*], Xiao Yu [a], Yan Xiao [b], Kwabena Ebo Bennin [c], Md Alamgir Kabir [a], Miao Zhang [a]

[a] Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong, China
[b] School of Computing, National University of Singapore, 117417, Singapore
[c] Information Technology Group, Wageningen University and Research, Wageningen, The Netherlands

ARTICLE INFO

Keywords:
Software defect prediction
Class imbalance
Oversampling
SMOTE
MAHAKIL
Effort-aware defect prediction

ABSTRACT

Context: Generally, there are more non-defective instances than defective instances in the datasets used for software defect prediction (SDP), which is referred to as the class imbalance problem. Oversampling techniques are frequently adopted to alleviate the problem by generating new synthetic defective instances. Existing techniques generate either near-duplicated instances which result in overgeneralization (high probability of false alarm, pf) or overly diverse instances which hurt the prediction model's ability to find defects (resulting in low probability of detection, pd). Furthermore, when existing oversampling techniques are applied in SDP, the effort needed to inspect the instances with different complexity is not taken into consideration.

Objective: In this study, we introduce Complexity-based OverSampling TEchnique (COSTE), a novel oversampling technique that can achieve low pf and high pd simultaneously. Meanwhile, COSTE also performs better in terms of $Norm(p_{opt})$ and ACC, two effort-aware measures that consider the testing effort.

Method: COSTE combines pairs of defective instances with similar complexity to generate synthetic instances, which improves the diversity within the data, maintains the ability of prediction models to find defects, and takes the different testing effort needed for different instances into consideration. We conduct experiments to compare COSTE with Synthetic Minority Oversampling TEchnique, Borderline-SMOTE, Majority Weighted Minority Oversampling TEchnique and MAHAKIL.

Results: The experimental results on 23 releases of 10 projects show that COSTE greatly improves the diversity of the synthetic instances without compromising the ability of prediction models to find defects. In addition, COSTE outperforms the other oversampling techniques under the same testing effort. The statistical analysis indicates that COSTE's ability to outperform the other oversampling techniques is significant under the statistical Wilcoxon rank sum test and Cliff's effect size.

Conclusion: COSTE is recommended as an efficient alternative to address the class imbalance problem in SDP.

1. Introduction

Models for software defect prediction (SDP) identify the defect-prone entities (e.g., files, packages, functions) in a software system [1]. An accurate prediction model helps developers focus on the predicted defects and utilize their time and effort effectively [2]. Many techniques have been proposed over the years [3–6] and applied to the development of software, helping practitioners allocate finite testing resources to those modules that tend to be defective [7–9]. In SDP, machine learning algorithms are the dominant approaches, which leverage historical data to train prediction models to predict defects [10–12]. Historical data are obtained from previous software projects, including the values of metrics (e.g. lines of code, depth of inheritance tree) and labels indicating whether the current instances are defective or non-defective.

By learning historical software information, prediction models gain knowledge about a software project and predict whether the instances introduced in future will be defective or not. However, in most projects, there are naturally many more non-defective instances (i.e., the majority class) than defective instances (i.e., the minority class), which is referred to as the class imbalance problem in SDP. In contrast,

* Corresponding authors.
E-mail addresses: shuofeng5-c@my.cityu.edu.hk (S. Feng), jacky.keung@cityu.edu.hk (J. Keung), xyu224-c@my.cityu.edu.hk (X. Yu), xiaoyan.hhu@gmail.com (Y. Xiao), kwabena.bennin@wur.nl (K.E. Bennin), makabir4-c@my.cityu.edu.hk (M.A. Kabir), miazhang9-c@my.cityu.edu.hk (M. Zhang).

https://doi.org/10.1016/j.infsof.2020.106432
Received 8 April 2020; Received in revised form 20 September 2020; Accepted 20 September 2020
Available online 25 September 2020

Abbreviations and acronyms

Abbreviation	Description
SDP	Software Defect Prediction
SQA	Software Quality Assurance
COSTE	Complexity-based Oversampling TEchnique
SMOTE	Synthetic Minority Oversampling TEchnique
MWMOTE	Majority Weighted Minority Oversampling TEchnique
ROS	Random OverSampling
RUS	Random UnderSampling
KNN	K-Nearest Neighbor
SVM	Support Vector Machine
RF	Random Forest
MLP	Multilayer Perceptron
DE	Differential Evolution algorithm
N	The population size of DE
G	The number of generations of DE
F	The mutation constant of DE
CR	The crossover rate of DE
pd	probability of detection
pf	probability of false alarms
ROC	Receiver Operating Characteristic
AUC	Area under the ROC curve

conventional machine learning algorithms assume that the numbers of minority class and majority class instances are roughly equal [13–15]. Prediction models trained by these highly imbalanced software datasets tend to ignore defective instances and to predict all outcomes as non-defective. As a result, the models produce highly biased results and cannot be applied in practice. Accordingly, developing techniques to effectively alleviate the class imbalance problem of software projects is a prevalent research topic in SDP, and several studies [12,16–18] have proposed different techniques or evaluated these techniques.

To tackle the class imbalance problem in SDP, data sampling techniques are widely adopted. The two general types of sampling techniques are oversampling and undersampling [19], the former of which generates new instances and adds them to the minority class, while the latter removes the existing instances from the majority class. Both techniques aim at balancing the distribution of the datasets to improve the performance of prediction models. In SDP, oversampling is preferable to undersampling, as instances discarded during the undersampling process may contain useful or important information for prediction models [20,21].

Random oversampling (ROS) is the most simple oversampling technique. It randomly replicates the minority class instances to balance the datasets. In contrast to ROS, most of the other existing oversampling techniques balance the imbalanced datasets by generating synthetic minority class instances. Synthetic Minority Oversampling TEchnique (SMOTE) [22] is the most common synthetic oversampling technique. It uses the K-nearest neighbor (KNN) algorithm to help select proper instances to generate synthetic minority class instances. Based on SMOTE, several other oversampling techniques, such as Borderline-SMOTE [23] and Adaptive Synthetic Sampling Approach (ADASYN) [24], have been developed. The reason that SMOTE adopts the KNN algorithm is based on the common assumption in the machine learning community that instances closer in distance are more similar than instances further in distance [25]. The synthetic minority class instances generated from the original minority class instances closer in distance are more likely to fall into the region of the minority class.

MAHAKIL [12], proposed by Bennin et al. is another synthetic oversampling technique. MAHAKIL computes the Mahalanobis distance [26] of the minority class instances and ranks the instances in the descending order based on that value. After being ranked, the ordered instances are partitioned into two bins based on the central instance. The two corresponding instances from each group are then averaged to generate synthetic instances.

However, the above techniques generate either near-duplicated instances (SMOTE-based oversampling techniques), which result in over-generalization (high probability of false alarm, pf) [27–29], or overly diverse instances (MAHAKIL), which hurt the prediction model's ability to find defects (resulting in low probability of detection, pd) [30].

In addition, most recent research [31–33] in SDP has taken into consideration the effort needed for data inspection, which is referred to as effort-aware defect prediction. For example, suppose there are two instances and each contains a defect. One instance is more complex than the other. The effort needed to inspect the complex instance could be much larger than the effort needed to inspect the less complex instance. Moreover, Zhang et al. [34], Nam et al. [35] and Menzies et al. [36] all pointed out that there is a strong relationship between the complexity and the defect-proneness of instances. Compared with the more complex defective instances, which are more likely to be correctly classified as defective, the less complex instances are easily misclassified as non-defective. Therefore, the less complex instances are likely to contain more useful information for prediction models and should be inspected first by the software quality assurance (SQA) team. However, the existing oversampling techniques in SDP neglect the variation in the complexity of instances.

In this study, we propose a novel oversampling technique, Complexity-based Oversampling TEchnique (COSTE), which can achieve low pf and high pd values simultaneously. Analogously to SMOTE, which assumes that the instances close in distance are similar, COSTE assumes that the instances close in complexity are similar and more likely to fall into the same region. Therefore, COSTE selects the instances that are similar in complexity instead of those that are close in distance to generate synthetic instances. By leveraging the complexity of instances to aid in instance selection, COSTE can generate diverse synthetic instances while still enhancing the ability of prediction models to find defects. COSTE first calculates the complexity of each instance and then ranks these instances based on complexity in the ascending order. Therefore, instances ranked adjacently have similar complexity while higher-ranked instances are less complex than those ranked below. COSTE uses higher-ranked instances more frequently than lower-ranked instances to generate synthetic instances, which makes the prediction models pay more attention to the less complex instances. It then averages pairs of instances ranked adjacently to generate synthetic instances, thus achieving a balanced dataset.

We conduct empirical experiments to compare the performance of COSTE to those of four common oversampling techniques (i.e., SMOTE, Borderline-SMOTE, MWMOTE and MAHAKIL). We use these oversampling techniques to oversample 23 datasets collected from the PROMISE repository [37]. Then the balanced datasets are used to train four common classifiers [i.e., support vector machine (SVM), K-nearest neighbor (KNN), random forest (RF) and multilayer perceptron (MLP)]. The experimental results show that COSTE improves the diversity of the synthetic instances (i.e., COSTE obtains lower pf) without compromising the ability of the prediction models to find defects (i.e., COSTE obtains higher pd) compared with the aforementioned four oversampling techniques. We also calculate $Norm(P_{opt})$ and ACC to evaluate the effort-aware performance of COSTE. Compared with the existing oversampling techniques, COSTE obtains better values of $Norm(P_{opt})$ and ACC, which indicates that it can find more defects than the other oversampling techniques with the same testing effort. Based on the Wilcoxon rank sum test (p-value <.05), the performance of COSTE is significantly better than the other four techniques.

Table 1
A simple example of defect prediction.

	LOC	NOC	WMC	Complexity
A	2	4	4	10
B	10	0	4	14
C	4	4	10	18
D	6	3	11	20

LOC, NOC and WMC are the abbreviations of Lines of Code, Number of Children and Weighted Method per Class respectively.

The rest of this paper is structured as follows: Section 2 introduces the motivation of our work. Section 3 presents the related work and background. Section 4 describes the details of COSTE. Section 5 explains the experimental design and Section 6 shows the experimental results. Section 7 analyzes the reasons for COSTE's superior performance. Section 8 discusses the threats to validity. Finally, Section 9 draws the conclusions and suggests future work.

2. Motivation

ROS replicates the minority class instances to balance the dataset. However, because ROS provides no new information for prediction models, overgeneralization is inevitable for prediction models trained by datasets that have been oversampled by ROS. SMOTE-based oversampling techniques balance the datasets by generating synthetic minority class instances. The generated minority class instances provide new information to train classifiers, unlike ROS. However, the diversity within the datasets is not significantly increased [12]. Due to the use of the KNN algorithm in SMOTE-based oversampling techniques, the instances that are used to generate synthetic instances are too close in distance, which may still result in the overgeneralization of prediction models. In addition, if the datasets contain several sub-clusters of the minority class, the use of the KNN algorithm will only generate synthetic instances that fall into a specific sub-cluster. These synthetic instances worsen the overgeneralization of prediction models.

To increase the diversity of oversampled datasets and address the overgeneralization of prediction models, Bennin et al. [12] proposed MAHAKIL, which selects pairs of unique and dissimilar instances to generate synthetic instances. Therefore, it is able to generate more diverse instances than SMOTE. However, according to Agrawal's work [30], although MAHAKIL increases the diversity of datasets, it hurts the prediction model's ability to find defects. In addition, combining two unique and dissimilar instances also raises the possibility that more erroneous synthetic instances will be introduced, which may negatively impact the performance of prediction models. Additionally, MAHAKIL adopts the Mahalanobis distance as its distance metric, which cannot be calculated for some datasets when the number of minority class instances is smaller than their dimensionality. In this case, to work properly, MAHAKIL needs to include multicollinearity techniques to eliminate some metrics.

To summarize, due to selecting instances that are too close in distance, SMOTE-based oversampling techniques generate near-duplicated and less diverse synthetic instances. MAHAKIL generates many synthetic minority class instances that wrongly fall outside the region of the minority class due to selecting dissimilar and unrelated instances, which hurts the ability of prediction models to find defects.

Based on our assumption that instances close in complexity are similar and more likely to fall into the same region, COSTE leverages the complexity of instances to aid in selecting those that are used to generate synthetic instances. There are three benefits to doing this. First, selecting instances close in complexity instead of distance can avoid generating near-duplicated instances as SMOTE-based oversampling techniques do, because instances similar in complexity do not have to be close in distance. For example, as observed in Table 1, instances A, B, C and D are the defective instances from a dataset. To take a simple example, the complexity of the instances is calculated as the sum of each metric with equal weight. The complexities of A, B and C are 10, 14 and 18, respectively. A is closer to B than C with regard to the complexity. However, the distance between A and B (8.94) is larger than that between A and C (6.32). This means the combination of A and B could produce a more diverse synthetic instance than the combination of A and C.

The second benefit is that the synthetic instances generated by two instances that are close in complexity also have similar complexity, which ensures that as few synthetic instances as possible are wrongly placed outside the region of the minority class, like in MAHAKIL, and thus enhances the ability of prediction models to find defects.

Furthermore, instances with different complexity have different tendencies to be defective in SDP. However, existing oversampling techniques do not consider the complexity of different instances. An extremely complex instance will have a very high probability of being defective, and thus contains little information for prediction models. In contrast, a less complex defective instance provides prediction models with more information, and therefore deserves more attention. In addition, considering the effort spent on inspecting instances, more defects could be found if the instances with less complexity are inspected first, assuming the same limits on testing capacity. COSTE therefore ranks instances in the ascending order based on the complexity of each. The higher-ranked instances are less complex than those ranked below. COSTE then uses the less complex instances to generate synthetic instances more often than those ranked below, which strengthens the less complex instances.

We also notice that existing oversampling techniques treat the weight of each metric identically, which is not realistic in practice. Therefore, we use a meta-heuristic method to automatically explore the optimal weight for each metric. Specifically, differential evolution (DE), which has exhibited good performance [38–40], is selected in this study. Once the optimal weight of each metric is fixed, we calculate the complexity of each instance by summing all weighted metrics and rank all instances by their complexity. Instances ranked adjacent have similar complexity. The adjacent instances are then averaged to generate synthetic instances. Using adjacent instances to generate synthetic instances enables COSTE to explore as many combinations of instances as possible, thus avoiding the case where generated synthetic instances fall into a certain sub-cluster like in SMOTE. The details of COSTE are presented in Section 4.

3. Related work and background

3.1. Class imbalance problem

The class imbalance problem is observed in various domains [41,42] and greatly degrades the performance of prediction models in SDP. Normally, there exist more non-defective than defective instances in the datasets used in SDP. Using these imbalanced datasets to train prediction models will bias the models toward the non-defective instances so that the trained models will tend to ignore the defective instances [43]. To alleviate the problem in SDP, many class imbalance learning techniques have been proposed. The three general techniques are data sampling [12,22,23], ensemble learning [44–47] and cost-sensitive learning [48–51].

Ensemble learning combines multiple classifiers and assigns different weights to each classifier. However, such techniques are normally time-consuming and still ignore the imbalance problem of datasets. In addition, although several prediction models can be combined in ensemble learning techniques, how to effectively leverage the combination of multiple prediction models and ensure the diversity of each individual model needs further investigation [50].

Cost-sensitive learning aims to build prediction models with the lowest misclassification cost. In SDP, misclassification of defective instances normally leads to higher cost than misclassification of non-defective instances [51]. However, assigning an appropriate cost to wrongly classified instances is an issue that has not been addressed.

We focus on data sampling techniques because they are independent of prediction models, easy to observe and faithfully represent the nature of the research object. In addition, data sampling is easier to apply and more practical than the two aforementioned techniques. As outlined above, undersampling and oversampling are the two general sampling techniques found in the literature [16,52,53].

For undersampling techniques, random undersampling (RUS) is the simplest form. It randomly removes instances from the majority class of the datasets to achieve dataset balance. However, undersampling is less common than oversampling in SDP, because the instances removed from the majority class may contain important information for classifiers. ROS, which randomly duplicates instances belonging to the minority class, is the simplest form of oversampling techniques. However, ROS only copies the existing minority class instances, which means it provides no new information for classifiers to learn and therefore may result in the overgeneralization of prediction models. To deal with this problem, Chawla et al. [22] proposed SMOTE, which generates new synthetic instances by combining a certain minority class instance with previously defined K minority class nearest neighbor instances. Because SMOTE generates synthetic instances by combining instead of replicating instances, it generates more diverse instances than ROS. However, SMOTE can still lead to the overgeneralization of prediction models because it only selects the nearest neighbor instances. On the basis of SMOTE, several modifications have been proposed (e.g., Borderline-SMOTE and MWMOTE). Borderline-SMOTE [23], proposed by Han et al. puts greater emphasis on those minority class instances that lie on the decision boundary. By selecting those instances, the decision boundary is strengthened. Barua et al. [54] proposed MWMOTE, which identifies the most informative minority class instances and leverages the information provided by the nearest majority class instances to assign different weights to those informative minority class instances. This enables the classifiers to emphasize those minority class instances that provide more information. Huda et al. [55] proposed a novel ensemble oversampling technique that combines several oversampling techniques to generate an ensemble prediction model. Lin et al. [56] proposed a novel oversampling technique, which is different from the idea of SMOTE. They extended the concept of Adaptive Subspace Self-organizing map to Kernel Adaptive Subspace Self-organizing map. Then the instances are generated in the well-trained subspaces and reconstructed in the original space.

MAHAKIL, proposed by Bennin et al. [12] aims to solve the overgeneralization problem brought by SMOTE and generate more diverse synthetic instances. It ranks instances based on the Mahalanobis distance and separates instances into two groups. The two corresponding instances from each group are together selected to generate new synthetic instances. The pairs of selected instances are not close in distance, which enhances the diversity of the synthetic instances. MAHAKIL outperforms and is more stable than SMOTE-based oversampling techniques. However, MAHAKIL fails to retain the ability to find defects when it improves the diversity of data, which makes it less useful. Moreover, another limitation of MAHAKIL is that the Mahalanobis distance cannot be calculated when the number of minority class instances is smaller than their dimensionality. Due to this limitation, MAHAKIL cannot function properly in some cases where the number of minority class instances is smaller than the number of metrics.

3.2. Effort-aware defect prediction

Although the conventional SDP prediction models based on binary classification algorithms achieve satisfactory performance, they do not distinguish between a complex and simple defects. The effort spent on inspecting a complex instance is normally much larger than that spent on inspecting a simple instance. However, testing resources are normally limited in reality, and it is impossible for SQA teams to inspect all instances in a software project. Therefore, effort-aware defect prediction has been proposed, which takes the effort spent on inspecting different instances into consideration. Kamei et al. [31] used lines of code as a proxy for effort and proposed a state-of-the-art effort-aware defect prediction model that could find 35% defects of a software project with only 20% of the effort that it would take to inspect all instances in the project. Compared with the conventional SDP prediction models, effort-aware prediction models are more useful in daily SQA activities and for aiding the development of high-quality software.

3.3. Differential evolution

Storn and Price [57] proposed DE as a method of real-parameter optimization, providing an alternative to inverting random bits. DE is similar to evolutionary algorithms (EAs) in that it is also based on mutation, crossover and selection to attain optimal parameters. Unlike conventional EAs, however, DE first evolves a population of candidate solutions and mutates the next generation by perturbing this generation with a scaled difference. Then, crossover is performed to improve the diversity of the population. By using a fitness function, the best candidate that has been seen in all generations is selected when DE terminates. This candidate is treated as the final solution. The brief stepwise procedure of DE is as follows:

(1) Initialization. DE initializes a population (at G = 0) that includes N randomly generated vectors $X_i^{(G)}$ with d features.

$$X_i^{(G)} = (x_{i,1}^{(G)}, x_{i,2}^{(G)}, \dots, x_{i,d}^{(G)}). \tag{1}$$

The size N of the population remains the same until the termination of DE. Each vector is a candidate solution in the population. The value of each feature in each vector should be in a certain range that is defined by the minimum bound as

$$X_{min} = (x_{min,1}, x_{min,2}, \dots, x_{min,d}), \tag{2}$$

and the maximum bound as

$$X_{max} = (x_{max,1}, x_{max,2}, \dots, x_{max,d}). \tag{3}$$

Normally, DE initializes the jth feature of the ith vector as follows

$$x_{i,j}^{(0)} = x_{min,j} + rand * (x_{max,j} - x_{min,j}), \tag{4}$$

where *rand* is a random number in the range from 0 to 1.

(2) Mutation. The mutant vector is generated after initialization according to

$$V_i^{(G+1)} = X_{r_1}^{(G)} + F * (X_{r_2}^{(G)} - X_{r_3}^{(G)}), \tag{5}$$

where r_1, r_2 and r_3 are the generated vectors and randomly selected from the population. F is a scaling factor between 0 and 2, which is used to control the amplification of $(X_{r_2}^{(G)} - X_{r_3}^{(G)})$.

(3) Crossover. After mutation, crossover is applied to increase the diversity of the mutant vector. The trial vector is introduced as

$$U_i^{(G+1)} = (u_{i,1}^{(G+1)}, u_{i,2}^{(G+1)}, \dots, u_{i,d}^{(G+1)}), \tag{6}$$

The crossover scheme of the feature $u_{i,j}^{(G+1)}$ is performed as

$$u_{i,j}^{(G+1)} = \begin{cases} v_{i,j}^{(G+1)} & \text{if } (randb(j) \le \text{CR}) or j = rnbr(i) \\ x_{i,j}^{(G)} & \text{if } (randb(j) > \text{CR}) and j \ne rnbr(i) \end{cases}. \tag{7}$$

In Eq. (7), *randb(j)* is a random number between 0 and 1. CR is the crossover rate, which is defined by the user in advance. $v_{i,j}^{(G+1)}$ is the jth feature of $V_i^{(G+1)}$. $rnbr(i)$ is a random number that is used to randomly choose the value of a feature $v_{i,j}^{(G+1)}$.

(4) Selection. The trial vector $U_i^{(G+1)}$ is selected as a new member of the next generation if the trial vector can maximize the fitness function. When DE terminates, the best candidate is selected as the final solution.

	Lines of Code	Number of Children	Weighted Methods per Class
X_1	10	0	4
X_2	2	4	4
X_3	6	3	11
X_4	4	4	10

Apply min-max normalization

	Lines of Code	Number of Children	Weighted Methods per Class
X_1	1	0	0
X_2	0	1.0	0
X_3	0.5	0.75	1.0
X_4	0.25	1.0	0.857

Fig. 1. Applying min–max normalization.

	Lines of Code	Number of Children	Weighted Methods per Class
X_1	1	0	0
X_2	0	1.0	0
X_3	0.5	0.75	1.0
X_4	0.25	1.0	0.857

Apply optimal weight to each metric

$\alpha_1 = 0.4$ $\alpha_2 = 0.6$ $\alpha_3 = 0.8$

	Lines of Code	Number of Children	Weighted Methods per Class
X_1	0.4	0	0
X_2	0	0.6	0
X_3	0.2	0.45	0.8
X_4	0.1	0.6	0.686

Fig. 2. Applying optimal weight.

4. Methodology

4.1. Overview of COSTE

In this study, we propose a novel oversampling method named Complexity-based OverSampling TEchnique (COSTE). In COSTE, we first calculate the weighted sum of each metric of an instance as its complexity. DE is applied to find the optimal weight for each metric. Next, COSTE ranks all instances in the ascending based on their complexity. The purpose of ranking is that the higher-ranked instances are less complex and should be inspected first in preference to the more complex instances. COSTE then averages pairs of adjacent minority defective instances to generate new synthetic instances. The detailed description of COSTE is introduced in the following subsections.

4.2. Applying min–max normalization

In this phase, the desired number of synthetic minority class instances is calculated first. Then we apply data normalization to the dataset to relieve the negative effect of different weights of various metrics on both the prediction models and the calculation of complexity. After applying data normalization, the metrics of all instances fall into a similar range of values, thus eliminating bias toward any particular metric. In this study, we adopt the min–max normalization method, which adjusts all values into the range from 0 to 1. The min–max normalization method is mathematically defined in (8)

$$X^* = \frac{(X - min)}{(max - min)}, \tag{8}$$

Table 2
Hyperparameter Configurations of DE.

Hyperparameters	Values
The population size, N	200
The number of generations, G	20
The mutation constant, F	0.3
The crossover rate, CR	0.9
The minimum bound, *min*	−1
The maximum bound, *max*	1

where X is the original value of the original data, and *min* and *max* represent the minimum and maximum values of the original data, respectively. X^* is the normalized value of the original data.

To illustrate the whole process more clearly, we take a simple example. Fig. 1 shows four defective instances X_1, X_2, X_3 and X_4 with the metrics of lines of code, number of children and weighted method per class. COSTE first applies the min–max normalization method to these four instances.

4.3. Calculating complexity and rank

Given a vector of metrics of an instance $X_i = (x_1, x_2, \ldots, x_d)$, we calculate the complexity of each instance based on Eq. (9)

$$Complexity_i = \sum_{j=1}^{d} \alpha_j x_j, \tag{9}$$

where d equals the number of metrics of an instance and α_j is the weight of each metric x_j.

In this study, α_j is optimized by DE. The range of α_j explored by DE is set from −1 to 1. If a certain metric is positively correlated with the complexity of instances, DE will automatically set it as a positive number. Otherwise, the value of this certain metric will be set as a negative number by DE. A fitness function is used to explore α_j so that the prediction models can achieve the best overall performance, which is measured by the area under the ROC curve (AUC) in this study (where ROC is the receiver operating characteristic). There is no uniform rule for the hyperparameter settings of DE. By convention, the population size is set to 10 times the dimensionality of instances. For the scaling factor F and the crossover rate CR, we tried different combinations. We chose values of F from 0.6 to 2.0 with intervals of 0.2. For CR, we chose values from 0.3 to 0.9 with intervals of 0.1. Experimentally, we found that the combination of F equaling 0.3 and CR equaling 0.9 minimized the prediction models' execution time while performing comparably to the other combinations. Therefore, we set F as 0.3 and CR as 0.9. The number of generations G is set to 20, because we found that convergence was achieved in no more than 20 generations on all four classifiers across all datasets. The detailed configuration of DE is presented in Table 2.

In Fig. 2, α_1, α_2 and α_3 are the optimal weights explored by DE for each metric.

Then the complexity of each instance can be computed by summing the values of each weighted metric. Having done this, COSTE ranks instances in the ascending order based on their complexity.

Fig. 3 shows the four instances with weighted metrics ranked in the ascending order based on their complexity, which is the weighted sum of each metric. X_1 is the instance with the lowest complexity, and X_3 is the instance with the highest complexity.

4.4. Generating new synthetic instances

In the final phase, COSTE generates new synthetic instances by averaging pairs of adjacent minority class instances from the top-ranked to the bottom-ranked. The complexity of each newly generated instance is the average complexity of the two adjacent instances. If the number of newly generated instances is still smaller than the desired number

	Lines of Code	Number of Children	Weighted Methods per Class
X_1	0.4	0	0
X_2	0	0.6	0
X_3	0.2	0.45	0.8
X_4	0.1	0.6	0.686

Calculate complexity and rank

	Lines of Code	Number of Children	Weighted Methods per Class	Complexity
X_1	0.4	0	0	0.4
X_2	0	0.6	0	0.6
X_4	0.1	0.6	0.686	1.386
X_3	0.2	0.45	0.8	1.45

Fig. 3. Calculating complexity and rank.

	Lines of Code	Number of Children	Weighted Methods per Class	Complexity
X_1	0.4	0	0	0.4
X_2	0	0.6	0	0.6
X_4	0.1	0.6	0.686	1.386
X_3	0.2	0.45	0.8	1.45

Generate synthetic instances

	Lines of Code	Number of Children	Weighted Methods per Class	Complexity
X_1	0.4	0	0	0.4
New_1	0.2	0.3	0	0.5
X_2	0	0.6	0	0.6
New_2	0.05	0.6	0.343	0.993
X_4	0.1	0.6	0.686	1.386
New_3	0.15	0.525	0.743	1.418
X_3	0.2	0.45	0.8	1.45

Fig. 4. Generating synthetic instances.

calculated previously, we insert the newly generated instances into the original dataset and repeat the above phases from 4.3 to 4.4 until the desired number is reached.

In Fig. 4, new synthetic instances are generated by averaging pairs of adjacent instances. For example, the synthetic instance New_1 is generated by averaging X_1 and X_2.

Algorithm 1 shows the pseudo-code of COSTE. Line 1 initializes an array for temporarily storing newly generated instances. Line 2 describes the phase of applying data normalization. Lines 3 and 4 are the phases of calculating complexity and ranking instances. In COSTE, n minority class instances can generate at most n-1 synthetic instances. If the needed number of synthetic instances is larger than the number of minority class instances minus 1, Lines 7 to 17 are executed. Otherwise, Lines 18 to 27 are executed. At the end, COSTE returns the balanced dataset.

Algorithm 1 COSTE algorithm

Input: Dataset N including the minority class instances N_{min} and the majority class instances N_{maj}
Output: Balanced dataset N_{bal}

1: $Array_{syn} \leftarrow$ array for storing new synthetic instances
2: apply the min-max normalization method to the dataset
3: for each instance X_i in N_{min}, calculate its complexity using Equation (9).
4: rank X_i in the ascending order based on complexity
5: calculate the number of new synthetic instances needed T, $T = number(N_{maj}) - number(N_{min})$
6: **if** $T > number(N_{min}) - 1$ **then**
7: **for** $i = 1, 2, ..., number(N_{min}) - 1$ **do**
8: new synthetic instance $X_{new} = (X_i + X_{(i+1)})/2$
9: add X_{new} into $Array_{syn}$
10: **end for**
11: update N_{min} by merging N_{min} and $Array_{syn}$
12: repeat Line 5
13: **else**
14: **for** $i = 1, 2, ..., T - 1$ **do**
15: new synthetic instance $X_{new} = (X_i + X_{(i+1)})/2$
16: add X_{new} into $Array_{syn}$
17: **end for**
18: update N_{min} by merging N_{min} and $Array_{syn}$
19: **end if**
20: **return** balanced dataset N_{bal} by merging N_{min} and N_{maj}

5. Experimental design

In this section, the research questions, the details of the datasets, the adopted oversampling techniques, the four classifiers, the performance measures, the statistical tests and the experimental procedure are presented.

5.1. Research questions

To evaluate COSTE, the following research questions are formulated.

RQ1: Does COSTE contribute to the diversity of the datasets?

SMOTE generates synthetic instances that are similar to the existing instances and thus lack diversity because only the nearest neighbor instances are selected, which may lead to overgeneralization of prediction models. To evaluate whether COSTE improves the diversity of the data distribution, we conduct experiments to compare the distance between the instances selected by oversampling techniques when applied separately to oversample datasets. In addition, according to Bennin [12], another indicator of the diversity of synthetic instances produced by an oversampling technique is a lower pf value. Therefore, we further compare the pf values of each oversampling technique. If the distance between the selected instances in COSTE is larger than those in SMOTE-based oversampling techniques and COSTE obtains a lower pf value, we can conclude that COSTE does contribute to the diversity within the data distribution.

RQ2: Does COSTE improve the diversity within the data distribution at the expense of its ability to find defects?

MAHAKIL [12] selects unique and dissimilar instances to generate synthetic minority class instances and thus improves the diversity within the data distribution. However, the improvement of the diversity is at the expense of the ability of prediction models to find defects, i.e., probability of detection (pd) values decrease. This makes MAHAKIL less useful than SMOTE-based oversampling techniques because it finds fewer defects. Whether COSTE improves the diversity within the data distribution at the expense of a decrease of pd values needs further investigation. Furthermore, when the testing resources are limited, it is preferable to select an oversampling technique that can find more defects with the same resources. $Norm(P_{opt})$ and ACC are commonly used in effort-aware defect prediction to reflect the ability of prediction models to find defects with limited testing resources. In this study, we obtain the values of $Norm(P_{opt})$ and ACC of each oversampling technique using 20% of the effort that it would take to inspect all instances. We thus conduct experiments and compare the resulting pd, $Norm(P_{opt})$ and ACC values of different oversampling techniques. If the values of pd, $Norm(P_{opt})$ and ACC of COSTE are comparable to or even better than SMOTE-based oversampling techniques and MAHAKIL, we can conclude that COSTE improves the diversity without compromising the ability of prediction models to find defects.

RQ3: How does COSTE perform compared with the existing oversampling techniques?

If COSTE successfully improves the diversity within the data distribution while maintaining the ability to find defects, its overall performance should be better than those of the other oversampling techniques. To compare their overall performances, we adopt AUC [58–60]

Table 3
Description of the metrics.

Abbreviation	Description
Static	
WMC	Weighted methods per class
DIT	Depth of Inheritance Tree
NOC	Number of children
CBO	Coupling between object classes
RFC	Response for a class
LCOM	Lack of cohesion in methods
CA	Afferent couplings
CE	Efferent couplings
NPM	Number of public methods
LCOM3	Lack of cohesion in methods, different from LCOM
LOC	Lines of code
DAM	Data access metric
MOA	Measure of aggregation
MFA	Measure of functional abstraction
CAM	Cohesion among methods of class
IC	Inheritance coupling
CBM	Coupling between methods
AMC	Average method complexity
MAX(CC)	Maximum value of CC methods of the investigated class
AVG(CC)	Arithmetic mean of the CC value in the investigated class

Table 4
Description of 23 imbalanced datasets collected from the PROMISE repository.

Projects	# Instances	Defect ratio
Ant-1.3	125	16
Ant-1.4	178	22.5
Ant-1.5	293	10.9
Ant-1.6	351	26.2
Ant-1.7	745	22.3
Camel-1.0	339	3.8
Camel-1.2	608	35.5
Camel-1.4	872	16.6
Camel-1.6	965	19.5
Ivy-1.4	241	6.6
Ivy-2.0	352	11.4
Jedit-3.2	272	33.1
Jedit-4.0	306	24.5
Jedit-4.1	312	25.3
Jedit-4.2	367	13.1
Log4j-1.0	135	25.2
Log4j-1.1	109	33.9
Poi-2.0	314	11.8
Synapse-1.2	256	33.6
Velocity-1.6	229	34.1
Xalan-2.4	723	15.2
Xerces-1.2	440	16.1
Xerces-1.3	453	15.2

and *balance* [16,61,62], which are commonly used as overall performance measures in SDP, where higher values would indicate that COSTE is superior to the other oversampling techniques

5.2. Datasets

To increase the generalizability of the experimental results, 23 releases of 10 open source projects from the PROMISE repository were included. These datasets are measured by the static metric. The percentages of defective instances in these selected projects are less than 50%. The details of the static metrics are presented in Table 3. The details of the adopted datasets are presented in Table 4.

5.3. Baselines

We compare COSTE with three common SMOTE-based oversampling techniques, SMOTE, Borderline-SMOTE and MWMOTE, and one recently proposed oversampling technique, MAHAKIL. The following is a brief introduction of these four techniques.

SMOTE. SMOTE [22] is the most common synthetic oversampling technique in SDP. Based on the KNN algorithm, SMOTE generates new synthetic instances by combining an instance with one of its K nearest neighbors, all of which belong to the minority class. The use of the KNN algorithm in SMOTE ensures that the newly generated instance lies in the region of the minority class. However, it also makes the performance of SMOTE heavily reliant on the selection of the nearest neighbor instances.

Borderline-SMOTE. Borderline-SMOTE [23] is an improved version of SMOTE. Instead of treating every instance equally, Borderline-SMOTE puts more focus on those instances that are hard for prediction models to classify. These instances are referred to as borderline instances because they are closer to the borderline between the majority and minority classes. This technique selects as the initial instances those borderline instances that have more majority class than minority class nearest neighbor instances. By doing this, the borderline between the majority and minority classes becomes clearer, thus improving the performance of prediction models.

MWMOTE. Different from SMOTE, which treats instances equally, MWMOTE [54] assigns different weights to those hard-to-learn minority class instances. MWMOTE leverages the information provided not only by the minority class instances, but also by the majority class instances. Based on the distance from the nearest majority class instances, MWMOTE calculates the different weights for different informative instances and then generate synthetic instances using a clustering approach.

MAHAKIL. Unlike SMOTE-based oversampling techniques, MAHAKIL [12] ranks and selects the minority class instances based on the Mahalanobis distance to generate new instances. In this way, it facilitates the selection of initial instances and reduces the overgeneralization problem, allowing it to outperform the previous oversampling techniques. However, due to the inherent limitations of the Mahalanobis distance, MAHAKIL is unreliable in situations when the number of minority class instances is smaller than their dimensionality. To deal with this limitation, MAHAKIL adopts the feature selection technique to reduce the dimensionality of instances when the number of minority class instances is small. To avoid introducing variation and bias and ensure an objective comparison among different oversampling techniques, we only select datasets with which MAHAKIL can work appropriately without applying the feature selection technique.

In this study, the K value for the KNN algorithm in SMOTE and Borderline-SMOTE is set to the default value of 5. As for the hyperparameter for MWMOTE, we follow the author's setting in [54]. The Euclidean distance is adopted as the distance metric in SMOTE, Borderline-SMOTE and MWMOTE.

5.4. Performance measures

The objective of SDP is to predict defective data as accurately as possible. However, when the datasets are imbalanced, the overall accuracy is not an appropriate performance measure. In such a situation, even if all instances were predicted as non-defective without any consideration of their actual properties, the overall accuracy would still remain high because the non-defective instances would vastly outnumber the defective instances. As such, performance measures that are not significantly affected by dataset imbalance are preferable, such as AUC and *balance*. In SDP, performance measures are normally computed based on the confusion matrix. A typical confusion matrix is shown in Table 5. Normally, defective instances are labeled as positive and non-defective instances as negative in SDP [63]. True positive (TP) represents the number of positive instances that are correctly predicted as positive. False positive (FP) represents the number of negative

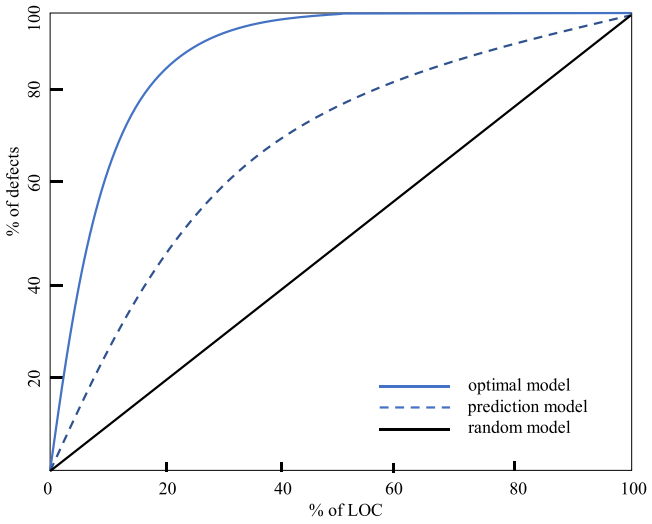

Fig. 5. An LOC-based cumulative lift chart.

instances are wrongly predicted as positive. True negative (TN) is the number of negative instances that are correctly predicted as negative. False negative (FN) is the number of positive instances that are wrongly predicted as negative. In this study, we adopt AUC, *balance*, *pd* and *pf* as the performance measures. These four indicators are widely used for performance evaluation [18,64,65] in SDP. AUC [66] and *balance* [36] are used to measure the overall performance of prediction models. *balance*, *pd* and *pf* are defined as below:

$$pd = recall = \frac{TP}{TP + FN} \tag{10}$$

$$pf = \frac{FP}{TN + FP} \tag{11}$$

$$balance = 1 - \frac{\sqrt{(0 - pf)^2 + (1 - pd)^2}}{\sqrt{2}} \tag{12}$$

For AUC, *balance* and *pd*, higher values represent better performance, while for *pf*, lower values are better. A high *balance* value indicates that *pd* is reasonably high and *pf* is reasonably low at the same time. Although we require a high *pd*, reflecting a strong ability to find defects, the trade-off between the two means that high *pd* values usually result in high *pf* values, which makes the model less efficient. The AUC metric can effectively describe the trade-off and better reflect the overall performance of a prediction model. Therefore, the fitness function in DE is designed to choose the solution that maximizes the AUC of prediction models. Then, we collect the values of *balance*, *pd* and *pf* when the maximum AUC is achieved.

In effort-aware defect prediction, the effort spent on inspecting instances is measured by their lines of code: the more lines of code an instance has, the more effort will be spent on inspecting it. $Norm(P_{opt})$ [32] and *ACC* [31] have been widely adopted to measure the effort-aware performance of prediction models in previous studies. *ACC* denotes the *pd* value when inspecting the top-ranked instances using 20% of the total effort that would be needed to inspect all instances. The mathematical definition of $Norm(P_{opt})$ is given below:

$$Norm(P_{opt}) = \frac{P_{opt} - worst(P_{opt})}{optimal(P_{opt}) - worst(P_{opt})} \tag{13}$$

In Eq. (13), P_{opt} is equal to $1 - \Delta_{opt}$, where Δ_{opt} is the area between the optimal model and the prediction model in the LOC-based cumulative lift chart. Fig. 5 presents an LOC-based cumulative lift chart.

Table 5
Confusion Matrix.

	Predicted positive	Predicted negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

5.5. Performance comparison

In this study, we compare COSTE with the other oversampling techniques with respect to diversity, ability to find defects and overall performance. Therefore, in addition to comparing the distance between the selected instances in each oversampling technique to check whether COSTE increases diversity, we compare the AUC, *balance*, *pd*, *pf*, $Norm(P_{opt})$ and *ACC* values of each technique to establish whether COSTE's performance is superior overall. We also adopt the Wilcoxon rank sum test (p-value <.05) to check whether the performance differences between any two oversampling techniques are significant. The Wilcoxon rank sum test is a non-parametric test that takes the null hypothesis that two techniques are the same while the alternative hypothesis is that two techniques are significantly different. If the p-value is less than 0.05, the null hypothesis is rejected, which indicates that there exists a statistically significant difference between the two techniques. When comparing two techniques, higher AUC, *balance*, *pd*, $Norm(P_{opt})$ and *ACC* values and lower *pf* values are better. Furthermore, to quantify the difference between the performance of COSTE and the other oversampling techniques beyond p-value interpretation, the effect size is computed (i.e., Cliff's δ). According to [67], we classify the effect size as negligible ($0 <$ Cliff's $\delta < 0.147$), small ($0.147 <$ Cliff's $\delta < 0.33$), medium ($0.33 <$ Cliff's $\delta < 0.474$) or large (Cliff's $\delta > 0.474$).

5.6. Classifiers

We adopt four common classifiers to enhance the generalizability of this study and the convenience of its replication by others, namely K-nearest neighbor (KNN) [68], random forest (RF) [69], support vector machine (SVM) [70] and multilayer perceptron (MLP) [71]. Because we focus on the performance of COSTE and the other oversampling techniques instead of tuning the hyperparameters of different classifiers, the sklearn package [72] in Python is adopted for our experiments and the hyperparameters for the selected classifiers are set to the default values.

5.7. Experimental procedure

To gain a comprehensive understanding of COSTE's performance, several experiments are conducted. First, we investigate the diversity of each oversampling technique to answer RQ1. For each oversampling technique, we calculate the Euclidean distance between the instances selected by the technique in a dataset. We conduct 10 replicates and calculate the average distance as the final result to reduce the variation and bias. In addition, considering Bennin's conclusion that low *pf* values represent high diversity of synthetic data, we also present the *pf* values of each oversampling technique. To compare the *pf* values of COSTE with those of the other oversampling techniques, we apply each oversampling technique to imbalanced datasets and build prediction models with the oversampled datasets. Before we apply each technique to the imbalanced datasets, we use 5-fold cross-validation to divide the datasets into five folds. To keep the ratio of minority class instances to majority class instances the same as in the original datasets, we use stratification to divide the datasets. Then we select four folds as the parent training data to build the prediction models, and the remaining fold is treated as the parent testing data to validate the models' performance. This is iterated five times to ensure all folds have been used for both training and testing. After the division is done, we apply COSTE and the other oversampling techniques only to the training data. When COSTE is applied, we further divide the

Table 6
Average distance between the selected instances of each oversampling technique.

Dataset	COSTE	SMOTE	Borderline	MWMOTE	MAHAKIL
ant-1.3	1.05	0.84	0.92	0.47	1.20
ant-1.4	0.85	0.66	0.63	0.53	1.41
ant-1.5	0.95	0.70	0.69	0.81	1.14
ant-1.6	0.83	0.54	0.67	1.05	1.11
ant-1.7	0.84	0.47	0.47	1.02	1.13
camel-1.0	1.13	1.04	0.92	1.05	1.33
camel-1.2	0.89	0.32	0.34	1.18	1.27
camel-1.4	0.95	0.40	0.40	1.13	1.19
camel-1.6	0.85	0.32	0.29	1.03	1.13
ivy-1.4	1.14	1.12	1.04	0.85	1.32
ivy-2.0	0.94	0.68	0.49	0.97	1.22
jedit-3.2	0.79	0.50	0.51	1.18	1.28
jedit-4.0	0.77	0.48	0.43	1.04	1.14
jedit-4.1	0.76	0.51	0.43	1.12	1.17
jedit-4.2	0.75	0.58	0.47	0.95	1.27
log4j-1.0	0.97	0.82	0.74	1.22	1.27
log4j-1.1	1.08	0.82	0.79	1.12	1.37
poi-2.0	0.96	0.65	0.49	0.97	1.18
synapse-1.2	1.11	0.59	0.74	1.17	1.30
velocity-1.6	0.95	0.51	0.50	1.13	1.27
xalan-2.4	1.01	0.56	0.57	1.02	1.36
xerces-1.2	0.79	0.46	0.48	0.74	1.49
xerces-1.3	0.98	0.57	0.59	1.12	1.39
average	0.93	0.61	0.59	0.99	1.26

four parent training folds into five subfolds. Then we take the four new subfolds as the training data and the remaining subfold as the testing data to decide the optimal weights for each metric by using DE. This procedure is also conducted five times to ensure all bins are used for training and testing. The parent testing data are kept unchanged. According to the assumption that most machine learning algorithms perform best when the data are balanced and the conclusion of Ahmad Abu [73] that oversampling techniques attain the best performance at 50%, we terminate the oversampling process when the numbers of minority class and majority class instances are equal. Then the balanced training data are used to train prediction models, and the pf values of these prediction models are validated by the testing data. We iterate the above process 10 times to reduce the impact of randomness. After 10 iterations, we take the average pf values of each technique as the final results. AUC, $balance$, pd, $Norm(P_{opt})$ and ACC are calculated in the same way as pf during the above process. Fig. 6 shows the whole flow of the experiments that compare each oversampling technique.

Based on the average distance and pf values, we can answer RQ1 and establish whether COSTE generates more diverse oversampled instances. To answer RQ2, we compare the pd, $Norm(P_{opt})$ and ACC values of each oversampling technique, with higher pd values indicating better ability to find defects and higher $Norm(P_{opt})$ and ACC values indicating that more defects can be found with the same testing effort. To answer RQ3, we compare the AUC and $balance$ values of each oversampling technique. If COSTE obtains higher AUC and $balance$ values than the other oversampling techniques, we can conclude that it is superior.

6. Experimental results

Here we exhibit the experimental results to answer the research questions, comparing the performance of each oversampling technique across each performance measure using the obtained performance values.

RQ1: Does COSTE contribute to the diversity of the datasets?

To answer RQ1, we first compare the average distance between the instances that are selected by each oversampling technique to generate synthetic instances. According to Bennin's work [12], the further the distance between those selected instances, the more diverse the distribution of the generated instances. Thus, we can inspect the diversity for each oversampling technique. Table 6 records the average distance for all 23 datasets oversampled by COSTE, SMOTE, Borderline-SMOTE (Borderline), MWMOTE and MAHAKIL. It can be clearly seen that MAHAKIL achieves the largest average distance (1.26), while the average distance of COSTE (0.93) is larger than those of SMOTE (0.61) and Borderline (0.59). Specifically, the distances of COSTE are larger than those of SMOTE and Borderline for all 23 individual datasets. The average distance of COSTE is similar to that of MWMOTE (0.99).

Table 7
Comparison among COSTE, SMOTE, Borderline, MWMOTE and MAHAKIL in terms of pf.

pf	COSTE	SMOTE	Borderline	MWMOTE	MAHAKIL
SVM	0.300	0.346	0.350	0.326	**0.293**
p-value		< **.05**	< **.05**	< **.05**	> .05
Cliff's δ		0.221	0.263	0.146	0.040
KNN	**0.214**	0.266	0.261	0.234	0.216
p-value		< **.05**	< **.05**	< **.05**	> .05
Cliff's δ		0.418	0.384	0.255	0.059
RF	0.126	0.128	0.128	**0.121**	0.129
p-value		> .05	> .05	> .05	> .05
Cliff's δ		0.028	0.025	0.059	0.028
MLP	**0.190**	0.222	0.229	0.208	0.191
p-value		< **.05**	< **.05**	< **.05**	> .05
Cliff's δ		0.395	0.448	0.259	0.041
average	**0.207**	0.241	0.242	0.223	0.207
p-value		< **.05**	< **.05**	< **.05**	> .05
Cliff's δ		0.353	0.289	0.183	0.028

According to Bennin [12], lower pf values represent more diverse synthetic instances. Therefore, we compare the pf values for each oversampling technique. From Table 7, we can see that COSTE obtains the lowest average pf value (0.207). MAHAKIL also performs well, with the same average pf value (0.207). Borderline performs the worst in terms of pf with a value of 0.242. Specifically, while COSTE obtains the lowest pf values on the KNN and MLP classifiers, it fails to do so on the SVM and RF classifiers, where instead MAHAKIL performs the best on the SVM classifier and MWMOTE performs the best on the RF classifier.

We also statistically analyze the pf values of each oversampling technique. Based on the Wilcoxon rank sum test at the confidence level of 95%, COSTE performs significantly better than SMOTE-based oversampling techniques in terms of average pf value. Specifically, COSTE is significantly better than SMOTE-based oversampling techniques on the SVM, KNN and MLP classifiers in terms of pf. The exception is that COSTE does not significantly outperform SMOTE-based oversampling techniques on the RF classifier. AS for the effect size, the differences between COSTE and SMOTE-based oversampling techniques are practically significant and that between COSTE and MAHAKIL is negligible in terms of pf.

To conclude the above observations, the diversity of COSTE is higher than those of SMOTE, Borderline and MWMOTE and is also comparable to that of MAHAKIL based on the average distance and pf values. This confirms our expectation that COSTE could generate more diverse synthetic instances than SMOTE-based oversampling techniques. Therefore, the answer to RQ1 is "yes": COSTE does contribute to the diversity within the data distribution.

RQ2: Does COSTE improve the diversity within the data distribution at the expense of its ability to find defects?

The ability of prediction models to find defects is reflected by pd, $Norm(P_{opt})$ and ACC, higher values of which represent higher defect-finding ability. Therefore, we compare pd, $Norm(P_{opt})$ and ACC values between different oversampling techniques to answer RQ2.

Table 8 records the pd values of all 23 datasets oversampled by COSTE, SMOTE, Borderline, MWMOTE and MAHAKIL on the SVM, KNN, RF and MLP classifiers as well as the p-values in terms of pd. We can see that all five oversampling techniques possess their highest pd values on the SVM classifier. COSTE achieves a better pd value (0.690)

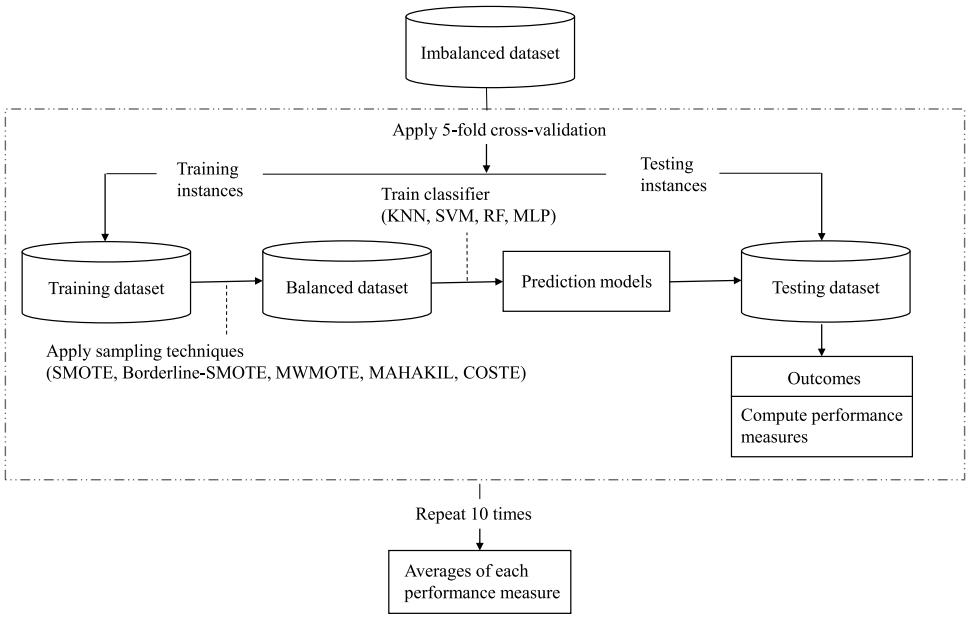

Fig. 6. Experimental framework of COSTE.

Table 8
Comparison among COSTE, SMOTE, Borderline, MWMOTE and MAHAKIL in terms of *pd*.

pd	COSTE	SMOTE	Borderline	MWMOTE	MAHAKIL
SVM	0.690	**0.692**	0.686	0.677	0.652
p-value		> .05	> .05	> .05	< **.05**
Cliff's δ		0.032	0.074	0.013	0.130
KNN	**0.623**	0.608	0.595	0.573	0.557
p-value		> .05	< **.05**	< **.05**	< **.05**
Cliff's δ		0.117	0.147	0.331	0.374
RF	**0.447**	0.432	0.430	0.432	0.433
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.100	0.110	0.108	0.079
MLP	0.595	0.596	**0.603**	0.582	0.568
p-value		> .05	> .05	> .05	< **.05**
Cliff's δ		0.040	0.085	0.021	0.062
average	0.589	0.583	0.578	0.566	0.552
p-value		>.05	> .05	< **.05**	< **.05**
Cliff's δ		0.043	0.025	0.130	0.168

Table 9
Comparison among COSTE, SMOTE, Borderline, MWMOTE and MAHAKIL in terms of $Norm(P_{opt})$.

$Norm(P_{opt})$	COSTE	SMOTE	Borderline	MWMOTE	MAHAKIL
SVM	**0.641**	0.632	0.628	0.630	0.621
p-value		> .05	> .05	> .05	< **.05**
Cliff's δ		0.040	0.070	0.051	0.081
KNN	**0.663**	0.642	0.639	0.630	0.621
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.214	0.244	0.353	0.365
RF	0.612	0.606	0.610	**0.613**	0.606
p-value		> .05	> .05	> .05	> .05
Cliff's δ		0.036	0.059	0.025	0.040
MLP	**0.633**	0.629	0.633	0.624	0.619
p-value		> .05	> .05	> .05	< **.05**
Cliff's δ		0.070	0.036	0.089	0.123
average	**0.637**	0.627	0.627	0.624	0.616
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.100	0.089	0.142	0.176

than Borderline, MWMOTE and MAHAKIL, but is inferior to SMOTE (0.692) on the SVM classifier. It can be seen that MAHAKIL obtains the lowest *pd* value among all techniques on that classifier. For the KNN classifier, COSTE achieves the best *pd* value (0.623) and MAHAKIL still performs the worst among all the techniques. For the RF classifier, COSTE gains the best *pd* value (0.447) while the other four techniques perform similarly. For the MLP classifier, Borderline obtains the best *pd* value (0.603). Regarding the average *pd* value, COSTE performs well, which shows that instead of decreasing the ability of finding defects, COSTE is good at finding defects as the other techniques, and MAHAKIL obtains a much lower average *pd* value than the other techniques. This is in agreement with Agrawal's finding [30] that MAHAKIL improves the diversity of the generated data at the expense of decreasing the ability to find defects.

Table 9 shows that COSTE outperforms all of the other oversampling techniques on the SVM, KNN and RF classifiers in terms of $Norm(P_{opt})$. Specifically, COSTE achieves a value of 0.641, while MAHAKIL performs the worst, with a value of only 0.621, on the SVM classifier. On the KNN classifier, the $Norm(P_{opt})$ value of COSTE is 0.663 and that of SMOTE is the second highest at 0.642. On the RF classifier, MAHAKIL performs the worst, with values of only 0.606, while COSTE still performs well and obtains 0.612 in terms of $Norm(P_{opt})$. On the MLP classifier, COSTE and Borderline performs the best. Regarding the average $Norm(P_{opt})$ value on all four classifiers, COSTE obtains 0.637 while MAHAKIL obtains the lowest average of only 0.616. Table 10 presents the *ACC* values of all five oversampling techniques. COSTE is the best-performing oversampling technique, outperforming all other techniques on all four classifiers in terms of *ACC*.

We again perform the Wilcoxon rank sum test at the confidence level of 95% to investigate the statistical significance of the differences between the compared oversampling techniques. From Table 8, we observe that COSTE significantly outperforms all the four techniques on the RF classifiers. In addition, COSTE performs significantly better than MAHAKIL on all the selected classifiers and also better than MWMOTE on the KNN classifier. From Table 9, the performance of COSTE is significantly better than those of SMOTE, Borderline, MWMOTE and MAHAKIL in terms of the average $Norm(P_{opt})$ values. COSTE also performs well on the KNN classifier, on which it significantly outperforms all the other techniques. In terms of the average *ACC* values, COSTE significantly outperforms all the compared techniques. Furthermore, the values of Cliff's δ effect sizes show that practical effect sizes exist between COSTE and MWMOTE and MAHAKIL in terms of *pd*, $Norm(P_{opt})$ and *ACC*. The Cliff's δ sizes between the performance of COSTE and those of SMOTE and Borderline are small.

Table 10
Comparison among COSTE, SMOTE, Borderline, MWMOTE and MAHAKIL in terms of *ACC*.

ACC	COSTE	SMOTE	Borderline	MWMOTE	MAHAKIL
SVM	**0.377**	0.358	0.352	0.355	0.355
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.074	0.123	0.089	0.119
KNN	**0.411**	0.380	0.374	0.365	0.361
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.206	0.227	0.267	0.259
RF	**0.347**	0.341	0.342	0.346	0.340
p-value		> .05	< **.05**	> .05	> .05
Cliff's δ		0.070	0.074	0.055	0.043
MLP	**0.385**	0.372	0.371	0.365	0.360
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.119	0.074	0.180	0.198
average	**0.380**	0.362	0.360	0.358	0.354
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.108	0.115	0.157	0.172

Table 11
Comparison among COSTE, SMOTE, Borderline, MWMOTE and MAHAKIL in terms of AUC.

AUC	COSTE	SMOTE	Borderline	MWMOTE	MAHAKIL
SVM	**0.692**	0.675	0.669	0.675	0.679
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.183	0.225	0.183	0.095
KNN	**0.704**	0.672	0.668	0.669	0.672
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.301	0.308	0.297	0.274
RF	**0.661**	0.654	0.653	0.655	0.654
p-value		< **.05**	< **.05**	< **.05**	> .05
Cliff's δ		0.066	0.104	0.062	0.070
MLP	**0.703**	0.688	0.687	0.687	0.688
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.130	0.172	0.104	0.089
average	**0.690**	0.672	0.669	0.672	0.674
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.191	0.191	0.172	0.153

Table 12
Comparison among COSTE, SMOTE, Borderline, MWMOTE and MAHAKIL in terms of *balance*.

balance	COSTE	SMOTE	Borderline	MWMOTE	MAHAKIL
SVM	**0.666**	0.647	0.641	0.643	0.652
p-value		< **.05**	< **.05**	< **.05**	> .05
Cliff's δ		0.180	0.233	0.202	0.089
KNN	**0.679**	0.652	0.644	0.642	0.641
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.229	0.323	0.312	0.285
RF	**0.593**	0.584	0.581	0.584	0.584
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.104	0.115	0.100	0.089
MLP	**0.673**	0.662	0.660	0.657	0.651
p-value		> .05	> .05	< **.05**	< **.05**
Cliff's δ		0.078	0.100	0.096	0.100
average	**0.653**	0.636	0.632	0.632	0.632
p-value		< **.05**	< **.05**	< **.05**	< **.05**
Cliff's δ		0.157	0.183	0.146	0.134

The value of *pd* is closely related to the value of *ACC*. Normally, the higher *pd* value, the higher *ACC* value. In this study, COSTE obtains similar *pd* values to SMOTE and Borderline while it obtains significantly better *ACC* values than SMOTE and Borderline, which highlights the positive effect of COSTE on effort-aware defect prediction.

To summarize, the performance of COSTE is comparable to or even better than those of the other techniques in terms of *pd*, $Norm(P_{opt})$ and *ACC* according to the experimental results and analysis above. Therefore, we can answer RQ2 by confirming that COSTE improves the diversity within the data distribution without degrading the ability of prediction models to find defects.

RQ3: How does COSTE perform compared to the existing oversampling techniques?

To answer RQ3, we leverage AUC and *balance* to compare the overall performance of COSTE and the other oversampling techniques. In [74], high *pd* and low *pf* are recommended as the more stable performance measures for oversampling techniques. As mentioned in regard to RQ1 and RQ2, COSTE achieves lower *pf* and higher *pd* values than the other oversampling techniques. If COSTE can also obtain higher AUC and *balance* values, we can conclude that it is superior to the other oversampling techniques.

Tables 11 and 12 show the AUC and *balance* values, respectively, of all five oversampling techniques on the four classifiers, together with the *p*-values. Table 11 shows that COSTE consistently outperforms all of the other oversampling techniques across all four classifiers in terms of AUC. COSTE with the KNN classifier achieves the highest AUC value (0.704). Considering its simplicity, the KNN classifier performs surprisingly well and holds its own as an accurate classifier. The average AUC value of COSTE is 0.690, which is much higher than those of SMOTE (0.672), Borderline (0.669), MWMOTE (0.672) and MAHAKIL (0.674). Similarly, COSTE also obtains higher *balance* values on all four classifiers than the other oversampling techniques, as shown in Table 12. Besides COSTE, the overall performances of the other four oversampling techniques are roughly equal in terms of both AUC and *balance* values.

We perform the Wilcoxon rank sum test at the confidence level of 95% to analyze whether the overall performance of COSTE is significantly better than those of the compared methods on the four classifiers over all datasets. From the results, it is obvious that the AUC values of COSTE are significantly higher than those of SMOTE, Borderline, MWMOTE and MAHAKIL on all classifiers. The only exception is that there is no significant difference between the performance of COSTE and that of MAHAKIL on the RF classifier. Considering *balance*, there is no significant difference between COSTE and MAHAKIL on the SVM classifier, and no significant difference between COSTE and SMOTE and Borderline on the MLP classifier. For the effect size, COSTE again outperforms the other oversampling techniques, achieving small or medium effect sizes in terms of average AUC and average *balance* values.

To show a more detailed comparison, we record the AUC values of each single dataset on the four classifiers in Tables 13–16. W/D/L in these tables represents Win/Draw/Loss, i.e., the number of the datasets on which COSTE performs better than, the same as, or worse than the other techniques in terms of AUC. These tables show that COSTE achieves the best AUC values and outperforms the other oversampling techniques on most datasets. The prediction models trained on the datasets oversampled by COSTE consistently have positive win-loss values. Specifically, COSTE wins on 18, 23 and 18 out of 23 datasets against SMOTE, Borderline and MWMOTE, respectively, on the SVM classifier. MAHAKIL performs well on that classifier, obtaining the second most wins. For the KNN classifier, COSTE strongly outperforms all of the other oversampling techniques on all datasets. On the RF and MLP classifiers, a similar trend is seen. MWMOTE gains the second most wins on the RF and MLP classifiers over the 23 datasets.

Fig. 7 shows the boxplot for AUC values of the 23 datasets oversampled by COSTE, SMOTE, Borderline, MWMOTE and MAHAKIL on the four classifiers. For the SVM classifier, COSTE achieves the highest median AUC value, SMOTE achieves the highest maximum and highest minimum AUC values. Moreover, COSTE gains the highest maximum, median and minimum AUC values on the KNN and RF classifiers. On the MLP classifier, COSTE obtains the highest maximum AUC value while Borderline obtains the highest median AUC value.

These experimental results show that COSTE achieves the best overall performance among the five oversampling techniques. Therefore, we can answer RQ3 by concluding that compared with the other oversampling techniques, COSTE performs better and obtains a more competitive prediction performance.

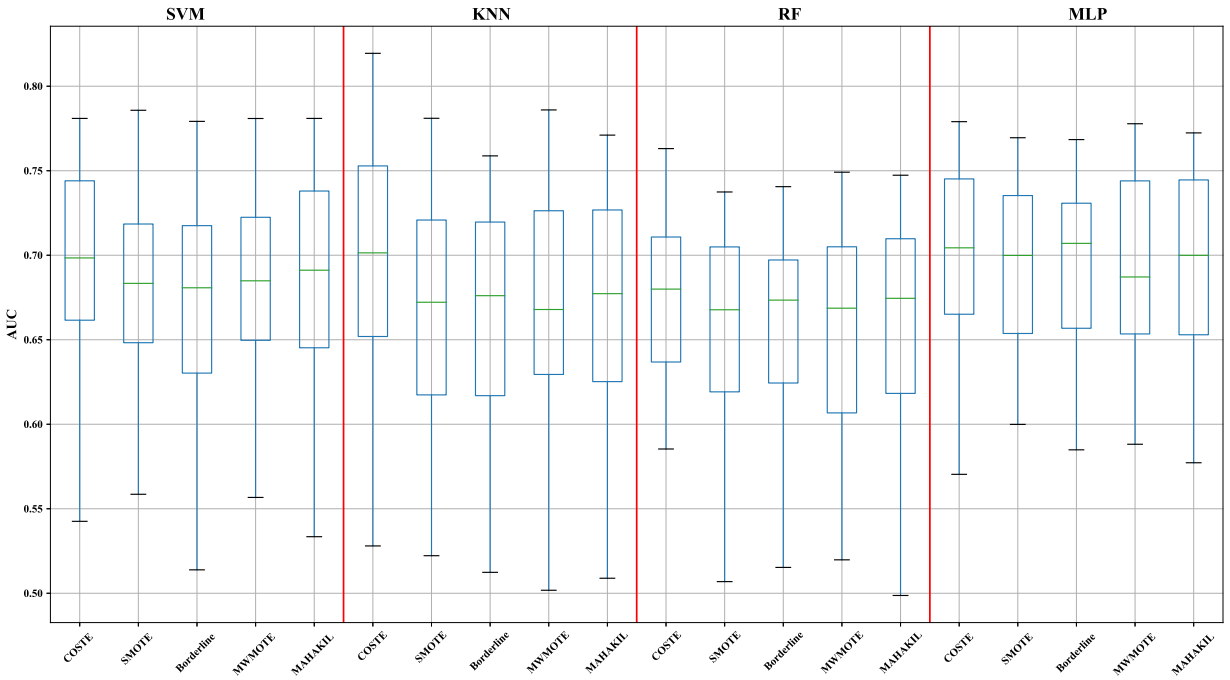

Fig. 7. Boxplot for AUC on 23 datasets oversampled by the five oversampling techniques on the four classifiers.

Table 13
AUC values on 23 datasets using the SVM classifier.

Dataset	COSTE	SMOTE	Borderline	MWMOTE	MAHAKIL
ant-1.3	0.758	0.749	0.740	0.734	0.781
ant-1.4	0.561	0.559	0.556	0.557	0.546
ant-1.5	0.728	0.688	0.680	0.698	0.700
ant-1.6	0.762	0.763	0.752	0.762	0.751
ant-1.7	0.722	0.715	0.701	0.719	0.726
camel-1.0	0.716	0.677	0.690	0.661	0.595
camel-1.2	0.543	0.536	0.521	0.525	0.533
camel-1.4	0.663	0.648	0.644	0.656	0.667
camel-1.6	0.599	0.588	0.586	0.593	0.590
ivy-1.4	0.662	0.627	0.588	0.617	0.642
ivy-2.0	0.738	0.711	0.718	0.733	0.715
jedit-3.2	0.766	0.734	0.715	0.751	0.752
jedit-4.0	0.694	0.697	0.692	0.695	0.693
jedit-4.1	0.746	0.689	0.676	0.679	0.705
jedit-4.2	0.743	0.721	0.734	0.724	0.759
log4j-1.0	0.744	0.720	0.732	0.719	0.742
log4j-1.1	0.781	0.786	0.779	0.781	0.780
poi-2.0	0.680	0.669	0.677	0.673	0.671
synapse-1.2	0.661	0.656	0.626	0.656	0.663
velocity-1.6	0.648	0.650	0.643	0.648	0.654
xalan-2.4	0.703	0.685	0.682	0.692	0.693
xerces-1.2	0.543	0.502	0.514	0.502	0.502
xerces-1.3	0.758	0.758	0.753	0.758	0.758
average	0.692	0.675	0.669	0.675	0.679
W/D/L		18/1/4	23/0/0	18/4/1	16/1/6

Table 14
AUC values on 23 datasets using the KNN classifier.

Dataset	COSTE	SMOTE	Borderline	MWMOTE	MAHAKIL
ant-1.3	0.712	0.658	0.686	0.661	0.702
ant-1.4	0.650	0.577	0.559	0.559	0.591
ant-1.5	0.755	0.727	0.692	0.725	0.728
ant-1.6	0.738	0.718	0.730	0.715	0.684
ant-1.7	0.745	0.722	0.720	0.727	0.736
camel-1.0	0.582	0.569	0.512	0.521	0.510
camel-1.2	0.570	0.552	0.553	0.530	0.549
camel-1.4	0.645	0.591	0.601	0.624	0.605
camel-1.6	0.670	0.633	0.635	0.633	0.622
ivy-1.4	0.528	0.522	0.520	0.502	0.509
ivy-2.0	0.726	0.669	0.689	0.682	0.707
jedit-3.2	0.762	0.741	0.759	0.733	0.743
jedit-4.0	0.763	0.750	0.748	0.749	0.736
jedit-4.1	0.767	0.716	0.719	0.738	0.732
jedit-4.2	0.747	0.702	0.701	0.716	0.724
log4j-1.0	0.784	0.723	0.731	0.734	0.716
log4j-1.1	0.819	0.781	0.755	0.786	0.771
poi-2.0	0.684	0.624	0.638	0.653	0.665
synapse-1.2	0.727	0.713	0.705	0.697	0.702
velocity-1.6	0.687	0.675	0.666	0.671	0.663
xalan-2.4	0.691	0.680	0.663	0.665	0.671
xerces-1.2	0.677	0.653	0.641	0.628	0.640
xerces-1.3	0.770	0.753	0.744	0.741	0.759
average	0.704	0.672	0.668	0.669	0.672
W/D/L		23/0/0	23/0/0	23/0/0	23/0/0

7. Discussion

7.1. Why COSTE performs the best

Essential to a superior oversampling technique is the ability to generate new instances that provide as much useful information as possible for prediction models to learn, while ensuring that as few newly generated instances as possible are wrongly introduced into the majority class. All SMOTE-based oversampling techniques share a similar strategy for generating synthetic instances, i.e., using the KNN algorithm to select the nearest neighbor instances for oversampling. This ensures that the generated instances correctly fall into the region of the minority class as far as possible. However, selecting instances that are close in distance risks the problem of generating instances that lack diversity, which leads to overgeneralization and the increase of pf values. In addition, if there are sub-clusters in the minority class instances, SMOTE-based oversampling techniques will only generate synthetic instances within each sub-cluster. This will worsen the overgeneralization of prediction models.

MAHAKIL uses a different strategy to select the instances that are used to generate synthetic instances, aiming at generating more diverse synthetic instances. By selecting pairs of dissimilar instances that are further in distance to ensure the diversity of the synthetic instances, it produces lower pf values than SMOTE-based oversampling techniques.

Table 15
AUC values on 23 datasets using the RF classifier.

Dataset	COSTE	SMOTE	Borderline	MWMOTE	MAHAKIL
ant-1.3	0.655	0.617	0.640	0.600	0.598
ant-1.4	0.606	0.600	0.594	0.596	0.614
ant-1.5	0.639	0.635	0.646	0.642	0.641
ant-1.6	0.733	0.737	0.731	0.735	0.747
ant-1.7	0.716	0.711	0.701	0.712	0.726
camel-1.0	0.496	0.517	0.515	0.520	0.499
camel-1.2	0.585	0.581	0.578	0.581	0.581
camel-1.4	0.611	0.613	0.609	0.599	0.596
camel-1.6	0.594	0.585	0.591	0.591	0.598
ivy-1.4	0.519	0.507	0.512	0.520	0.511
ivy-2.0	0.644	0.644	0.634	0.648	0.635
jedit-3.2	0.737	0.736	0.737	0.735	0.730
jedit-4.0	0.712	0.695	0.694	0.700	0.699
jedit-4.1	0.746	0.725	0.728	0.735	0.720
jedit-4.2	0.693	0.701	0.698	0.696	0.689
log4j-1.0	0.701	0.661	0.675	0.707	0.692
log4j-1.1	0.763	0.737	0.741	0.749	0.730
poi-2.0	0.636	0.627	0.621	0.625	0.630
synapse-1.2	0.722	0.720	0.715	0.723	0.713
velocity-1.6	0.680	0.702	0.689	0.685	0.678
xalan-2.4	0.642	0.633	0.634	0.635	0.641
xerces-1.2	0.655	0.661	0.652	0.652	0.664
xerces-1.3	0.706	0.706	0.683	0.688	0.720
average	0.661	0.654	0.653	0.655	0.654
W/D/L		15/2/6	18/1/4	14/0/9	15/0/8

Table 16
AUC values on 23 datasets using the MLP classifier.

Dataset	COSTE	SMOTE	Borderline	MWMOTE	MAHAKIL
ant-1.3	0.743	0.681	0.710	0.670	0.703
ant-1.4	0.635	0.602	0.609	0.611	0.641
ant-1.5	0.673	0.659	0.678	0.652	0.667
ant-1.6	0.766	0.764	0.762	0.768	0.761
ant-1.7	0.763	0.749	0.731	0.743	0.749
camel-1.0	0.684	0.618	0.585	0.616	0.599
camel-1.2	0.602	0.600	0.596	0.607	0.604
camel-1.4	0.662	0.661	0.665	0.671	0.654
camel-1.6	0.652	0.652	0.651	0.657	0.623
ivy-1.4	0.570	0.518	0.526	0.522	0.505
ivy-2.0	0.728	0.729	0.733	0.718	0.706
jedit-3.2	0.779	0.765	0.768	0.772	0.772
jedit-4.0	0.744	0.725	0.714	0.713	0.747
jedit-4.1	0.772	0.760	0.755	0.744	0.752
jedit-4.2	0.746	0.737	0.732	0.734	0.727
log4j-1.0	0.748	0.729	0.731	0.714	0.738
log4j-1.1	0.771	0.770	0.765	0.764	0.772
poi-2.0	0.645	0.633	0.656	0.640	0.653
synapse-1.2	0.736	0.721	0.721	0.719	0.731
velocity-1.6	0.696	0.700	0.704	0.713	0.696
xalan-2.4	0.698	0.700	0.694	0.704	0.696
xerces-1.2	0.604	0.606	0.597	0.622	0.577
xerces-1.3	0.743	0.751	0.722	0.756	0.758
average	0.703	0.688	0.687	0.687	0.688
W/D/L		17/1/5	18/0/5	15/0/8	16/1/6

However, the use of such distantly separated instances may accidentally widen the minority class boundary, so that the synthetic instances may wrongly fall outside the region of the minority class. This degrades the ability of prediction models to find defects and results in low *pd* values.

COSTE leverages the complexity of instances to aid in selecting those that are used to generate synthetic instances. This avoids the issues mentioned above. First, COSTE can avoid selecting instances that are too close together because instances with similar complexity do not have to be close in distance. In addition, using adjacent instances to generate synthetic instances, COSTE can explore more possible combinations of instances and avoid all of the generated synthetic instances falling into a certain sub-cluster, like in SMOTE. Therefore, COSTE generates more diverse synthetic instances than SMOTE-based oversampling techniques. Second, to generate synthetic instances, COSTE selects pairs of related instances rather than dissimilar instances like MAHAKIL, so that no synthetic instance falls outside the region of the minority class, thus protecting the ability of the prediction models to find defects.

Table 17
Comparison between COSTE and D-COSTE.

Classifier	Technique	AUC	$Norm(P_{opt})$	ACC
SVM	COSTE	0.692	0.641	0.377
	D-COSTE	0.689	0.636	0.373
	p-value	> .05	> .05	> .05
KNN	COSTE	0.704	0.663	0.411
	D-COSTE	0.703	0.657	0.406
	p-value	> .05	> .05	> .05
RF	COSTE	0.661	0.612	0.347
	D-COSTE	0.654	0.609	0.331
	p-value	> .05	> .05	< **.05**
MLP	COSTE	0.703	0.633	0.385
	D-COSTE	0.700	0.630	0.377
	p-value	> .05	> .05	< **.05**

In addition, SMOTE-based oversampling techniques and MAHAKIL consider all metrics of instances as equal in weight, which is not realistic, as different metrics make different contributions to the likelihood of discovering a defect. COSTE uses DE to optimize the weight for each metric. Varying the weight of different metrics forces the prediction models to pay more attention to metrics of higher weight and therefore learn more useful information, which enhances their performance.

7.2. Do instances ranked in the descending or ascending order really affect the performance of coste?

In COSTE, we rank instances in the ascending order based on complexity for two reasons: (1) the defective instances of lower complexity provide the prediction models with more information than those of higher complexity and (2) the less complex instances should be inspected first to minimize the testing effort. In COSTE, the higher-ranked instances will be used more often to generate synthetic instances than the instances ranked below them. To prove that ranking instances in the ascending order rather than the descending order does affect the performance of the prediction models, we rerun COSTE under the setting that ranks instances in the descending order, referred to as D-COSTE. Table 17 presents the comparison between COSTE and D-COSTE. We can see that the performance of COSTE is consistently better than that of D-COSTE in terms of AUC, $Norm(P_{opt})$ and ACC on all four classifiers. On the SVM and KNN classifiers, the difference between the performance of COSTE and that of D-COSTE is not significant. However, on the RF classifier, COSTE significantly outperforms D-COSTE in terms of ACC. On the MLP classifier, the ACC value of COSTE is significantly better than that of D-COSTE. To summarize, COSTE consistently outperforms D-COSTE, which agrees with our assumption as well as justifies our choice to rank instances in the ascending order in COSTE.

7.3. The contribution of different metrics to the complexity of instances

In this section, we discuss the optimal weight of each metric explored by DE. We believe that these optimal weights can reflect the contribution of different metrics to the complexity of instances to some extent. As mentioned in Section 5.7, we adopt 5-fold cross-validation and iterate 10 times for each dataset. Therefore, we will get 50 results for each dataset. We select the best performance of COSTE from the 50 results in terms of AUC. We do this because when the performance of COSTE achieves the best, the optimal weight could better reflect the actual weight of metrics. To investigate which metric contributes more to the complexity of instances, we rank all the metrics by the magnitude of the absolute values of the optimal weights. Because of

Table 18
The five most influential metrics for the datasets achieving the highest AUC values on the SVM classifier.

	Metric 1	Metric 2	Metric 3	Metric 4	Metric 5
ant-1.6	CAM +	IC +	DIT +	NOC −	MFA −
jedit-3.2	CBM −	CA −	WMC −	NPM +	LOC +
jedit-4.1	NPM +	AVG(CC) −	AMC +	CBO −	LCOM3 −
log4j-1.1	CBM +	LCOM +	LCOM3 −	CAM +	MAX(CC) −

+ indicates positively correlated metrics.
− indicates negatively correlated metrics.

Table 19
The five most influential metrics for the datasets achieving the highest AUC values on the KNN classifier.

	Metric 1	Metric 2	Metric 3	Metric 4	Metric 5
ant-1.6	NOC +	MFA −	NPM −	AVG(CC) +	WMC −
jedit-3.2	CBM +	MFA −	DIT −	LCOM −	DAM −
jedit-4.1	MOA +	IC −	CE −	RFC +	AVG(CC) +
log4j-1.1	MAX(CC) −	WMC −	LCOM +	CAM −	CBO +

+ indicates positively correlated metrics.
− indicates negatively correlated metrics.

Table 20
The five most influential metrics for the datasets achieving the highest AUC values on the RF classifier.

	Metric 1	Metric 2	Metric 3	Metric 4	Metric 5
ant-1.6	CA +	NOC −	LCOM −	DIT +	DAM −
jedit-3.2	NPM −	RFC −	DIT +	LOC −	AMC −
jedit-4.1	IC +	CE +	WMC −	AMC −	LCOM −
log4j-1.1	NOC +	NPM −	MFA −	CE −	LOC +

+ indicates positively correlated metrics.
− indicates negatively correlated metrics.

the limited space, we only present the correlation of the five most influential metrics with the complexity of instances, and part of the studied datasets, on which COSTE achieves the highest AUC values among all the studied datasets.

From Tables 18 to 21, we can see that the metrics of DIT, LCOM, CE and CAM are the most influential metrics to the complexity of an instance based on the number of occurrences of each metric. However, for each single dataset on each classifier, the most influential metrics vary a lot. For example, the five most influential metrics for ant-1.6 on the SVM classifier is CAM, IC, DIT, NOC and MFA, which is totally different from that of jedit-3.2 on the SVM classifier. Therefore, we conclude that the distributions of different datasets in SDP vary significantly and the future oversampling techniques should pay more attention to the differences among different metrics to improve.

8. Threats to validity

This section discusses the threats to the external, internal, and construct validity of our experimental study.

Table 21
The five most influential metrics for the datasets achieving the highest AUC values on the MLP classifier.

	Metric 1	Metric 2	Metric 3	Metric 4	Metric 5
ant-1.6	LOC +	CE +	LCOM3 −	NOC −	LCOM +
jedit-3.2	CE +	CAM −	AMC −	RFC −	AVG(CC) +
jedit-4.1	CA −	CAM +	LCOM3 +	DIT +	RFC +
log4j-1.1	RFC −	CAM −	DIT −	CE −	LCOM3 −

+ indicates positively correlated metrics.
− indicates negatively correlated metrics.

External Validity. In our study, the experiments were performed on 23 datasets. These datasets were selected from the PROMISE repository and have been used in several previous studies [12,65,75]. Here, only static code metrics were used in our experiments. We cannot claim that our results are generalizable to other types of metrics such as process metrics. The lack of generalizability to other datasets or other types of metrics threatens the external validity of our results. However, static code metrics were widely adopted and have performed well in previous studies [76–78], and datasets measured by static code metrics are easy to collect. Moreover, a detailed description of our technique and the experiments are provided. Therefore, it would be easy to replicate our experiments using any available dataset with different types of metrics. In addition, only four classifiers were adopted in our experiments and the parameters were set to the defaults. The performance of our proposed technique on other classifiers or with different parameters was not validated here. Four common oversampling techniques were selected for comparison. We intend to extend our comparison to more oversampling techniques and more classifiers in the future.

Internal Validity. COSTE depends on DE to find the optimal weight of each metric. DE has several advantages such as few parameter settings, high performance, and applicability to high-dimensional complex optimization problems. However, the unstable convergence is a drawback of DE. In addition, the parameters in SMOTE-based oversampling techniques are randomly generated, which could also have introduced some bias into our results. To minimize such bias, we iterated the experiments 10 times to reduce the randomness and instability. In addition, the original MAHAKIL adopts multicollinearity techniques to eliminate some metrics when the number of defective instances is smaller than the dimensionality of the minority class instances. This is necessary because the Mahalanobis distance cannot be computed when the number of minority class instances is less than the dimensionality. To ensure an objective comparison between the oversampling techniques, we only selected datasets whose defective instances were more numerous than their dimensionality. With these datasets, MAHAKIL can work properly without applying multicollinearity techniques.

Construct Validity. One threshold-independent performance measure (AUC) and five threshold-dependent performance measures ($balance$, pd, pf, $Norm(P_{opt})$ and ACC) were used to evaluate the performance of these oversampling techniques. These six selected performance measures are common in SDP. However, if other performance measures are adopted, different results may be obtained. To reach a more general conclusion, we plan to adopt more performance measures in our future work.

9. Conclusion and future work

In SDP, defect-containing datasets are normally imbalanced, which is referred to as the class imbalance problem. Oversampling techniques are the common choice to alleviate the problem. However,

existing oversampling techniques generate either near-duplicated instances, which result in overgeneralization and high pf, or overly diverse instances, which hurt the ability of the prediction models to find defects and result in low pd. In addition, existing oversampling techniques do not take into consideration the fact that the effort needed for inspecting different instances varies considerably. To alleviate these issues, we propose a novel oversampling technique named Complexity-based Oversampling TEchnique (COSTE). COSTE leverages the complexity of instances, instead of the distance between them, to aid in selecting those that are used to generate synthetic instances. COSTE can avoid selecting instances that are too close in distance and instead generate more diverse instances by selecting the those that are similar in complexity. Additionally, by selecting instances close in complexity, it avoids selecting those that are too dissimilar, which would hurt the ability of the prediction models to find defects. Moreover, COSTE forces the prediction models to pay more attention to the less complex instances by prioritizing these during synthetic instance generation.

We empirically evaluate the effect of COSTE by comparison with four oversampling techniques (SMOTE, Borderline-SMOTE, MWMOTE and MAHAKIL) using four classifiers (SVM, KNN, RF and MLP) on 23 imbalanced datasets. The experimental results show that COSTE consistently performs significantly better than the other four techniques in terms of all performance measures (AUC, $balance$, pd, pf, $Norm(P_{opt})$ and ACC) at the confidence level of 95%. Specifically, COSTE significantly outperforms SMOTE-based oversampling techniques and is comparable to MAHAKIL in terms of pf. COSTE decreases pf by 16.9% compared to SMOTE-based oversampling techniques. COSTE also outperforms both SMOTE based oversampling techniques and MAHAKIL in terms of pd. COSTE increases pd by 6.7% compared to MAHAKIL. The performance of COSTE is also significantly better than those of SMOTE-based oversampling techniques and MAHAKIL when measured using overall metrics. COSTE increases AUC and $balance$ by 3.1% and 3.3%. Considering the effort needed to inspect instances, COSTE increases $Norm(P_{opt})$ and ACC by 3.4% and 7.3% respectively. Due to the superior performance of COSTE, we recommend it as an efficient alternative to address the class imbalance problem in SDP.

In our future work, we intend to generalize COSTE to more software datasets with various types of metrics, more classifiers and more performance measures to validate the technique's generalizability. Besides, we also plan to develop a framework based on the complexity of instances to improve the overall performance of effort-aware defect prediction models. Moreover, considering that COSTE uses DE to optimize the weights of the metrics of instances, which increases the execution time, we plan to adopt various techniques such as parallel techniques [6] to expedite the execution of COSTE. By leveraging the information provided by the complexity of instances, we plan to further extend our work to semi-supervised or unsupervised learning research similar to the work of Huda et al. [79].

CRediT authorship contribution statement

Shuo Feng: Conceptualization, Methodology, Experiment, Writing. **Jacky Keung:** Supervision. **Xiao Yu:** Conceptualization, Writing - review & editing. **Yan Xiao:** Writing - review & editing. **Kwabena Ebo Bennin:** Writing - review & editing. **Md Alamgir Kabir:** Writing - review & editing. **Miao Zhang:** Writing - review & editing.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.infsof.2020.106432.

Acknowledgments

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (No. 11208017) and the research funds of City University of Hong Kong (7005028, 7005217), and the Research Support Fund by Intel, China (9220097), and funding supports from other industry partners (9678149, 9440227, 9440180 and 9220103).

References

[1] F.J. Buckley, R. Poston, Software quality assurance, IEEE Trans. Softw. Eng. SE-10 (1) (1984) 36–41, http://dx.doi.org/10.1109/TSE.1984.5010196.
[2] F. Zhang, A.E. Hassan, S. McIntosh, Y. Zou, The use of summation to aggregate software metrics hinders the performance of defect prediction models, IEEE Trans. Softw. Eng. 43 (5) (2016) 476–491.
[3] N. Limsettho, K.E. Bennin, J.W. Keung, H. Hata, K. Matsumoto, Cross project defect prediction using class distribution estimation and oversampling, Inf. Softw. Technol. 100 (2018) 87–102.
[4] Z. Wan, X. Xia, A.E. Hassan, D. Lo, J. Yin, X. Yang, Perceptions, expectations, and challenges in defect prediction, IEEE Trans. Softw. Eng. (2018).
[5] Z. Li, X.-Y. Jing, F. Wu, X. Zhu, B. Xu, S. Ying, Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction, Autom. Softw. Eng. 25 (2) (2018) 201–245.
[6] M.M. Ali, S. Huda, J. Abawajy, S. Alyahya, H. Al-Dossari, J. Yearwood, A parallel framework for software defect detection and metric selection on cloud computing, Cluster Comput. 20 (3) (2017) 2267–2281.
[7] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in: Proceedings of the 28th International Conference on Software Engineering, ACM, 2006, pp. 452–461.
[8] T.J. Ostrand, E.J. Weyuker, R.M. Bell, Predicting the location and number of faults in large software systems, IEEE Trans. Softw. Eng. 31 (4) (2005) 340–355.
[9] P. Tomaszewski, H. Grahn, L. Lundberg, A method for an accurate early prediction of faults in modified classes, in: 2006 22nd IEEE International Conference on Software Maintenance, IEEE, 2006, pp. 487–496.
[10] Y. Ma, G. Luo, X. Zeng, A. Chen, Transfer learning for cross-company software defect prediction, Inf. Softw. Technol. 54 (3) (2012) 248–256.
[11] A. Okutan, O.T. Yıldız, Software defect prediction using Bayesian networks, Empir. Softw. Eng. 19 (1) (2014) 154–181.
[12] K.E. Bennin, J. Keung, P. Phannachitta, A. Monden, S. Mensah, MAHAKIL: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction, IEEE Trans. Softw. Eng. 44 (6) (2018) 534–550, http://dx.doi.org/10.1109/TSE.2017.2731766.
[13] B. Krawczyk, Learning from imbalanced data: open challenges and future directions, Prog. Artif. Intell. 5 (4) (2016) 221–232.
[14] G.M. Weiss, F. Provost, The effect of class distribution on classifier learning: an empirical study, 2001.
[15] K. Yoon, S. Kwek, A data reduction approach for resolving the imbalanced data issue in functional genomics, Neural Comput. Appl. 16 (3) (2007) 295–306.
[16] K.E. Bennin, J.W. Keung, A. Monden, On the relative value of data resampling approaches for software defect prediction, Empir. Softw. Eng. 24 (2) (2019) 602–636, http://dx.doi.org/10.1007/s10664-018-9633-6.
[17] K.E. Bennin, J. Keung, A. Monden, Impact of the distribution parameter of data sampling approaches on software defect prediction models, in: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2017, pp. 630–635.
[18] L. Chen, B. Fang, Z. Shang, Y. Tang, Tackling class overlap and imbalance problems in software defect prediction, Softw. Qual. J. 26 (1) (2018) 97–125.
[19] X. Zhang, Q. Song, G. Wang, K. Zhang, L. He, X. Jia, A dissimilarity-based imbalance data classification algorithm, Appl. Intell. 42 (3) (2015) 544–565.
[20] L. Zhou, Performance of corporate bankruptcy prediction models on imbalanced dataset: The effect of sampling methods, Knowl.-Based Syst. 41 (2013) 16–25.
[21] N. Japkowicz, S. Stephen, The class imbalance problem: A systematic study, Intell. Data Anal. 6 (5) (2002) 429–449.
[22] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, SMOTE: synthetic minority over-sampling technique, J. Artificial Intelligence Res. 16 (2002) 321–357.
[23] H. Han, W.-Y. Wang, B.-H. Mao, Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning, in: International Conference on Intelligent Computing, Springer, 2005, pp. 878–887.
[24] H. He, Y. Bai, E.A. Garcia, S. Li, ADASYN: Adaptive synthetic sampling approach for imbalanced learning, in: 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence), IEEE, 2008, pp. 1322–1328.
[25] D.M.J. Tax, One-class classification: Concept learning in the absence of counter-examples, 2002.
[26] R. De Maesschalck, D. Jouan-Rimbaud, D.L. Massart, The mahalanobis distance, Chemometr. Intell. Lab. Syst. 50 (1) (2000) 1–18.

[27] G.Y. Wong, F.H. Leung, S.-H. Ling, A novel evolutionary preprocessing method based on over-sampling and under-sampling for imbalanced datasets, in: Iecon 2013-39th Annual Conference of the Ieee Industrial Electronics Society, IEEE, 2013, pp. 2354–2359.

[28] B. Turhan, T. Menzies, A.B. Bener, J. Di Stefano, On the relative value of cross-company and within-company data for defect prediction, Empir. Softw. Eng. 14 (5) (2009) 540–578.

[29] K.E. Bennin, J. Keung, A. Monden, P. Phannachitta, S. Mensah, The significant effects of data sampling approaches on software defect prioritization and classification, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2017, pp. 364–373.

[30] A. Agrawal, T. Menzies, Is "better data" better than "better data miners"?, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 1050–1061.

[31] Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, N. Ubayashi, A large-scale empirical study of just-in-time quality assurance, IEEE Trans. Softw. Eng. 39 (6) (2012) 757–773.

[32] T. Mende, R. Koschke, Effort-aware defect prediction models, in: 2010 14th European Conference on Software Maintenance and Reengineering, IEEE, 2010, pp. 107–116.

[33] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, H. Leung, Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 157–168.

[34] F. Zhang, Q. Zheng, Y. Zou, A.E. Hassan, Cross-project defect prediction using a connectivity-based unsupervised classifier, in: Proceedings of the 38th International Conference on Software Engineering, ACM, 2016, pp. 309–320.

[35] J. Nam, S. Kim, CLAMI: Defect prediction on unlabeled datasets (T), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 452–463, http://dx.doi.org/10.1109/ASE.2015.56.

[36] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Trans. Softw. Eng. 33 (1) (2006) 2–13.

[37] J.S. Shirabad, T.J. Menzies, The PROMISE Repository of Software Engineering Databases, vol. 24, School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[38] P.P. Biswas, P.N. Suganthan, R. Mallipeddi, G.A. Amaratunga, Optimal power flow solutions using differential evolution algorithm integrated with effective constraint handling techniques, Eng. Appl. Artif. Intell. 68 (2018) 81–100.

[39] F. Zhang, C. Deb, S.E. Lee, J. Yang, K.W. Shah, Time series forecasting for building energy consumption using weighted support vector regression with differential evolution optimization technique, Energy Build. 126 (2016) 94–103.

[40] A. Onan, S. Korukoğlu, H. Bulut, A multiobjective weighted voting ensemble classifier based on differential evolution algorithm for text sentiment classification, Expert Syst. Appl. 62 (2016) 1–16.

[41] Y. Zhang, J.-x. Li, J. Zhao, S.-z. Wang, Y. Pan, K. Tanaka, S. Kadota, Synthesis and activity of oleanolic acid derivatives, a novel class of inhibitors of osteoclast formation, Bioorganic Med. Chem. Lett. 15 (6) (2005) 1629–1632.

[42] M.D. Saçar, J. Allmer, Data mining for microrna gene prediction: on the impact of class imbalance and feature number for microrna gene prediction, in: 2013 8th International Symposium on Health Informatics and Bioinformatics, IEEE, 2013, pp. 1–6.

[43] F. Provost, Machine learning from imbalanced data sets 101, in: Proceedings of the AAAI'2000 Workshop on Imbalanced Data Sets, vol. 68, AAAI Press, 2000, pp. 1–3.

[44] Z. Sun, Q. Song, X. Zhu, Using coding-based ensemble learning to improve software defect prediction, IEEE Trans. Syst. Man Cybern. B 42 (6) (2012) 1806–1817, http://dx.doi.org/10.1109/TSMCC.2012.2226152.

[45] I.H. Laradji, M. Alshayeb, L. Ghouti, Software defect prediction using ensemble learning on selected features, Inf. Softw. Technol. 58 (2015) 388–402, http://dx.doi.org/10.1016/j.infsof.2014.07.005, URL http://www.sciencedirect.com/science/article/pii/S0950584914001591.

[46] X. Xia, D. Lo, E. Shihab, X. Wang, X. Yang, ELBlocker: Predicting blocking bugs with ensemble imbalance learning, Inf. Softw. Technol. 61 (2015) 93–106, http://dx.doi.org/10.1016/j.infsof.2014.12.006, URL http://www.sciencedirect.com/science/article/pii/S0950584914002602.

[47] H. Wang, T.M. Khoshgoftaar, A. Napolitano, A comparative study of ensemble feature selection techniques for software defect prediction, in: 2010 Ninth International Conference on Machine Learning and Applications, 2010, pp. 135–140, http://dx.doi.org/10.1109/ICMLA.2010.27.

[48] M. Liu, L. Miao, D. Zhang, Two-stage cost-sensitive learning for software defect prediction, IEEE Trans. Reliab. 63 (2) (2014) 676–686, http://dx.doi.org/10.1109/TR.2014.2316951.

[49] 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, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 414–423, http://dx.doi.org/10.1145/2568225.2568320, URL http://doi.acm.org/10.1145/2568225.2568320.

[50] X. Yu, M. Wu, Y. Jian, K.E. Bennin, M. Fu, C. Ma, Cross-company defect prediction via semi-supervised clustering-based data filtering and MSTrA-based transfer learning, Soft Comput. 22 (10) (2018) 3461–3472, http://dx.doi.org/10.1007/s00500-018-3093-1.

[51] D. Tomar, S. Agarwal, Prediction of defective software modules using class imbalance learning, Appl. Comp. Intell. Soft Comput. 2016 (2016) 6:6, http://dx.doi.org/10.1155/2016/7658207.

[52] C. Drummond, R.C. Holte, et al., C4. 5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling, in: Workshop on Learning from Imbalanced Datasets II, vol. 11, Citeseer, 2003, pp. 1–8.

[53] X. Guo, Y. Yin, C. Dong, G. Yang, G. Zhou, On the class imbalance problem, in: 2008 Fourth International Conference on Natural Computation, Vol. 4, IEEE, 2008, pp. 192–201.

[54] S. Barua, M.M. Islam, X. Yao, K. Murase, MWMOTE–majority weighted minority oversampling technique for imbalanced data set learning, IEEE Trans. Knowl. Data Eng. 26 (2) (2012) 405–425.

[55] S. Huda, K. Liu, M. Abdelrazek, A. Ibrahim, S. Alyahya, H. Al-Dossari, S. Ahmad, An ensemble oversampling model for class imbalance problem in software defect prediction, IEEE Access 6 (2018) 24184–24195.

[56] C.-T. Lin, T.-Y. Hsieh, Y.-T. Liu, Y.-Y. Lin, C.-N. Fang, Y.-K. Wang, G. Yen, N.R. Pal, C.-H. Chuang, Minority oversampling in kernel adaptive subspaces for class imbalanced datasets, IEEE Trans. Knowl. Data Eng. 30 (5) (2017) 950–962.

[57] R. Storn, K. Price, Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces, J. Global Optim. 11 (4) (1997) 341–359.

[58] L. Gong, S. Jiang, R. Wang, L. Jiang, Empirical evaluation of the impact of class overlap on software defect prediction, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 698–709.

[59] T. Zhou, X. Sun, X. Xia, B. Li, X. Chen, Improving defect prediction with deep forest, Inf. Softw. Technol. 114 (2019) 204–216.

[60] N. Li, M. Shepperd, Y. Guo, A systematic review of unsupervised learning techniques for software defect prediction, Inf. Softw. Technol. (2020) 106287.

[61] Z. Li, X.-Y. Jing, X. Zhu, H. Zhang, B. Xu, S. Ying, Heterogeneous defect prediction with two-stage ensemble learning, Autom. Softw. Eng. 26 (3) (2019) 599–651.

[62] X. Xia, D. Lo, S.J. Pan, N. Nagappan, X. Wang, Hydra: Massively compositional model for cross-project defect prediction, IEEE Trans. Softw. Eng. 42 (10) (2016) 977–998.

[63] Y. Jiang, B. Cukic, Y. Ma, Techniques for evaluating fault prediction models, Empir. Softw. Eng. 13 (5) (2008) 561–595, http://dx.doi.org/10.1007/s10664-008-9079-3.

[64] T. Maciejewski, J. Stefanowski, Local neighbourhood extension of SMOTE for mining imbalanced data, in: 2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM), IEEE, 2011, pp. 104–111.

[65] P. He, B. Li, X. Liu, J. Chen, Y. Ma, An empirical study on software defect prediction with a simplified metric set, Inf. Softw. Technol. 59 (2015) 170–190.

[66] A.P. Bradley, The use of the area under the ROC curve in the evaluation of machine learning algorithms, Pattern Recognit. 30 (7) (1997) 1145–1159.

[67] V.B. Kampenes, T. Dybå, J.E. Hannay, D.I. Sjøberg, A systematic review of effect size in software engineering experiments, Inf. Softw. Technol. 49 (11–12) (2007) 1073–1086.

[68] T. Cover, P. Hart, Nearest neighbor pattern classification, IEEE Trans. Inform. Theory 13 (1) (1967) 21–27.

[69] Y. Ma, L. Guo, B. Cukic, A statistical framework for the prediction of fault-proneness, in: Advances in Machine Learning Applications in Software Engineering, IGI Global, 2007, pp. 237–263.

[70] F. Xing, P. Guo, M.R. Lyu, A novel method for early software quality prediction based on support vector machine, in: 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05), IEEE, 2005, pp. 10–pp.

[71] S. Sharmeen, S. Huda, J. Abawajy, M.M. Hassan, An adaptive framework against android privilege escalation threats using deep learning and semi-supervised approaches, Appl. Soft Comput. 89 (2020) 106089.

[72] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., Scikit-learn: Machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.

[73] A.A. Shanab, T.M. Khoshgoftaar, R. Wald, A. Napolitano, Impact of noise and data sampling on stability of feature ranking techniques for biological datasets, in: 2012 IEEE 13th International Conference on Information Reuse Integration (IRI), 2012, pp. 415–422, http://dx.doi.org/10.1109/IRI.2012.6303039.

[74] T. Menzies, A. Dekhtyar, J. Distefano, J. Greenwald, Problems with precision: A response to "comments on'data mining static code attributes to learn defect predictors'", IEEE Trans. Softw. Eng. 33 (9) (2007) 637–640.

[75] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, K. Matsumoto, An empirical comparison of model validation techniques for defect prediction models, IEEE Trans. Softw. Eng. 43 (1) (2016) 1–18.

[76] G. Fan, X. Diao, H. Yu, K. Yang, L. Chen, Software defect prediction via attention-based recurrent neural network, Sci. Program. 2019 (2019).

[77] M.M. Öztürk, Which type of metrics are useful to deal with class imbalance in software defect prediction? Inf. Softw. Technol. 92 (2017) 17–29.

[78] J. Li, P. He, J. Zhu, M.R. Lyu, Software defect prediction via convolutional neural network, in: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2017, pp. 318–328.

[79] S. Huda, S. Miah, M.M. Hassan, R. Islam, J. Yearwood, M. Alrubaian, A. Almogren, Defending unknown attacks on cyber-physical systems by semi-supervised approach and available unlabeled data, Inform. Sci. 379 (2017) 211–228.