TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 04/18 pp41–57
DOI: 10.26599/TST.2020.9010040
Volume 27, Number 1, February 2022

A Novel Cross-Project Software Defect Prediction Algorithm Based on Transfer Learning

Shiqi Tang, Song Huang*, Changyou Zheng*, Erhu Liu, Cheng Zong, and Yixian Ding

Abstract: Software Defect Prediction (SDP) technology is an effective tool for improving software system quality that has attracted much attention in recent years. However, the prediction of cross-project data remains a challenge for the traditional SDP method due to the different distributions of the training and testing datasets. Another major difficulty is the class imbalance issue that must be addressed in Cross-Project Defect Prediction (CPDP). In this work, we propose a transfer-leaning algorithm (TSboostDF) that considers both knowledge transfer and class imbalance for CPDP. The experimental results demonstrate that the performance achieved by TSboostDF is better than those of existing CPDP methods.

Key words: Software Defect Prediction (SDP); transfer learning; imbalance class; cross-project

1 Introduction

With the increasing scale, the structure of a software system becomes increasingly complex. Software defects occur for a number of reasons, such as a misunderstanding of the software requirements, an ineffective development process, or a lack of software development experience. Generally, defective softwares result in unanticipated economic losses to enterprises and even human-related costs[1]. If software defects can be found during the early stage of development, the software quality can be improved and its related cost is reduced. Thus, the rapid early detection of defects during the software development process is a challenging and important task for both software developers and testers.

Methods based on machine learning are feasible approaches for predicting software defects[2–6]. A common practice with these methods is to build a prediction model by mining the local data of software repositories. To date, traditional machine-learning algorithms, including decision trees, naive Bayes, neural networks, logistic regression, and Support Vector Machines (SVMs), have been successfully implemented by many researchers for predicting defects[7, 8]. K. O. Elish and M. O. Elish[9] compared the performance of eight machine-learning algorithms on the National Aeronautics Space Administration (NASA) dataset, and the results showed that SVMs performed better than the other reference algorithms. Studies have reported improvements in the traditional machine-learning methods by consideration of the specific characteristics of Software Defect Prediction (SDP). For example, Jing et al.[10] proposed a dictionary-learning-based SDP algorithm known as Cost-sensitive Discriminative Dictionary Learning (CDDL), and demonstrated that better performance could be achieved on the NASA dataset by CDDL than SVM. Compressed C45 Models (CCM) algorithm proposed by Wang et al.[11] and an ant colony optimization SVM algorithm proposed by Jiang et al.[12] also achieved better performance than traditional machine-learning algorithms. Li et al.[13] and Lu et al.[14] labeled some modules in software projects based on the concepts of active learning and semi-supervised learning, to help models predict defects in software.

• Shiqi Tang, Song Huang, Changyou Zheng, Erhu Liu, and Cheng Zong are with Command & Control Engineering College, Army Engineering University of PLA, Nanjing 210000, China. E-mail: tttpgs@163.com; hs0317@163.com; zheng_chy@163.com; waterworld625@126.com; zongcheng@ cnnp.com.cn.
• Yixian Ding is with Foreign Language College, Liaoning Technical University, Fuxin 123000, China, E-mail: 13904187373@163.com.
* To whom correspondence should be addressed.
Manuscript received: 2020-08-28; accepted: 2020-09-18

Unlike traditional machine-learning cases, SDP datasets typically suffer from class imbalance. Two approaches have been proposed to overcome this problem. One is the use of random oversampling and the random undersampling/synthetic-minority oversampling technique to preprocess data, which arranges label distributions of the data[15, 16]. The other approach is the application of cost-sensitive learning and ensemble learning technologies to modify the training process, which achieves high precision in the prediction of minority class samples.

Most of the above researches have focused on within-project defect prediction. However, sometimes, few local training data are available in the target software in which defects must be detected, because past local defective modules are expensive to label or the module development occurs in an unfamiliar domain[17]. To solve this challenge, one viable solution is to utilize data from other projects, known as source projects, to train a model for use in detecting defects within the target project. This is known as Cross-Project Defect Prediction (CPDP). However, previous studies have found that due to the different distributions of data between the source and target projects, traditional defect prediction methods have difficulty in achieving good performance in CPDP. Transfer learning is a machine-learning technology that has attracted a tremendous amount of attention in recent years[18, 19], because it utilizes source domain knowledge for model learning in the target domain when the target domain has a limited amount of labeled data. Recently, transfer-learning-based methods were proposed for CPDP. For example, Nam et al.[20] proposed the TCA+ algorithm, Ma et al.[17] proposed the transfer naive Bayes algorithm, and Chen et al.[21] proposed a double transfer boosting algorithm. However, the challenge of class imbalance in CPDP data remains. In this paper, we propose a CPDP method, TSboostDF, that combines weight-based sampling and transfer learning. Compared with other SDP methods, TSboostDF achieves better performance on CPDP datasets.

2 Background

2.1 Software defect prediction

In the software development process, software defects are inevitably introduced by inexperienced developers or developers who misunderstand the software requirements, which lead to high repair costs. The cost of detecting and repairing defects increases significantly once the software is released, so early detection is required before releasing software to the open market. However, a thorough examination and detection of the defects in all modules involve large amounts of manpower and resources. Thus, project managers must identify program modules that may contain defects in advance and assign sufficient testing resources to them.

SDP technology based on machine-learning and historical development data is considered to be a feasible solution to this challenge and is typically applied to predict the defect proneness of the program modules in software projects. Jing et al.[10] proposed CDDL algorithm, based on the supervised dictionary-learning technique for mining defect distribution information in historical data, and reported that it exhibited better performance than traditional machine-learning algorithms. The CCM algorithm proposed by Wang et al.[11] was based on C4.5 and Spearman's rank correlation coefficient for predicting software defects. As noted above, Jiang et al.[12] reported good SDP using a combination of SVM and ant colony optimization. In addition, many researchers have applied the latest research advances in machine-learning to SDP, such as semi-supervised learning and active learning. For example, Lu et al.[14] constructed a defect prediction model based on active learning. They not only used data from previous software version, but also used a small number of labeled data from the current software version to train the model. Li et al.[13] proposed the CoForest and ACoForest algorithms. CoForest is a sampling method based on semi-supervised learning. Using random forest classification, this method finds optimal sampling examples by random sampling. ACoForest extends the performance of CoForest via active learning. Studies have reported that these two methods perform better than other traditional machine-learning methods.

2.2 Class imbalances in software defect prediction

Class imbalance is a normal occurrence in SDP. Chen et al.[1] reported that class imbalance can be attributed to the distributions of defects in software modules, which is roughly consistent with the Pareto principles (80% of defects are concentrated in 20% of the program modules). Therefore, the number of non-defect-prone moduli (i.e., majority class) is larger than that of the defect-prone moduli (i.e., minority class), which leads to low prediction accuracy for defect-prone modules. This not only reduces the quality

of the software (if the classifier predicts a defect-prone module as a non-defect-prone module, it will generate defects in the final-release software), but also increases the cost of software development (if a classifier predicts a non-defect-prone module as a defect-prone module, significant amounts of time and manpower will be spent on testing this module, which wastes test resources). There are two feasible approaches for solving the above challenges. One is the application of class-distribution-based methods, such as random oversampling, random undersampling, and synthetic-minority oversampling techniques in the adjustment of the label distribution. For example, Menzies et al.[16] used a random sampling method to preprocess data and found that removing some samples from the training data does not significantly reduce the performance of the defect prediction model. Pelayo and Dick[15] also showed that data preprocessing by synthetic-minority oversampling effectively improves the performance of the model. The second approach is based on modification of the learning algorithm, in which better prediction accuracy is achieved in the minority class by modifying the algorithm training process. Typical methods include cost-sensitive learning and ensemble learning. Zheng[22] proposed a cost-sensitive algorithm based on neural networks to overcome the impact of class imbalance. Wang and Yao[23] compared the performances of several machine-learning algorithms to find a suitable method for SDP. The experimental results revealed that the ensemble learning algorithm, AdaBoost.NC, exhibites better performance than other algorithms.

2.3 Cross-project defect prediction based on transfer learning

To date, most researches have focused on within-project defect prediction, whereby a defect prediction model is constructed based on part of the data in the project (local training data) with other data being used to evaluate the model performance. However, local training data can rarely be used in practice for a number of reasons, such as the expense associated with labeling local defective modules or developing fresh module domains for a project. To solve this problem, the CPDP method uses a source-project data training model to detect defects in the target project. Zimmermann et al.[24] constructed 622 cross-project task combinations from 12 projects to evaluate CPDP model performance. Interestingly, the experimental results showed that traditional defect prediction methods have difficulty achieving good CPDP performance, because they assumed that the training and test data have similar distributions, whereas data from different projects have different distributions. Transfer learning, which has attracted increasing attention in recent years[18, 19], utilizes source domain information to promote model learning when the target domain contains limited labeled data. Transfer learning has achieved good performances in many domains, including document classification[25], machine translation[26], visual brain computer[27], and data stream classification[28]. Ma et al.[17] considered CPDP belonging in the transfer-learning domain. In recent years, several methods based on transfer learning have been proposed for CPDP. Turhan et al.[29] reported a selection method based on nearest-neighbor sample and used this method to select source-project data that are similar to the target project data for classifier training. Nam et al.[20] proposed a method called TCA+, which applies transfer component analysis[30] to explore the latent feature space among different project data. When the potential space is determined, the source and target domains are mapped to the space, which effectively eliminates the impact of differences in the data distributions of various projects. Ma et al.[17] assigned different weights to data from each source project based on their degree of similarity with data from the target project, and proposed a weighted naive Bayes model, known as TNB. In a previous study, Chen et al.[21] reported a double-transfer boosting algorithm, which has two stages—data gravitation and transfer boosting. Data gravitation is used to reconstruct the distribution of features in the source project based on the target project data, and transfer boosting is applied to remove negative instances in the source project. Li et al.[31] used the multi-source transfer learning algorithm for predicting cross-project software defects by transferring data information from multiple source projects to the target project. However, the class imbalance in the training data is neglected in these transfer-learning algorithms, which can affect their performances.

To solve the above problems, we propose a new CPDP algorithm, TSboostDF. We constructed 42 CPDP tasks based on the Promise dataset, and then compared the performances of TSboostDF with other SDP methods on a CPDP dataset. The experimental results demonstrated that TSboostDF achieved outstanding prediction performance.

3 Approach

In this section, TSboostDF is described in detail. The construction of TSboostDF has three stages—initialization, training, and prediction. In the initialization stage, we assign two weights, a sample weight and a similar weight for each dataset in the source project. In the training stage, we use the boosting method to train several base classifiers. Before each base classifier is established, the training and test sets are reconstructed based on the sample weights, then these datasets are used to build the basic classifiers and calculate their weights. In the prediction stage, these classifiers are used by TSboostDF to predict defects.

3.1 Problem description

In this paper, We define the source project and the target project as $\boldsymbol{D}_S = \{(\boldsymbol{X}_i^S,\ y_i^S)|i = 1, 2, \ldots, \mathrm{Sn}\}$ and $\boldsymbol{D}_T = \{\boldsymbol{X}_k^T|k = 1, 2, \ldots, \mathrm{Tn}\}$, respectively, where $\boldsymbol{X}_i^S$ and $\boldsymbol{X}_k^T \in \mathbf{R}^{m\times 1}$, Sn is the number of samples in the source-project and Tn is the number of samples in the target project. Note that (1) $\boldsymbol{X}_i^S$ and $\boldsymbol{X}_k^T$ represent the feature vector of i-th sample and the k-th sample in $\boldsymbol{D}_S$ and $\boldsymbol{D}_T$, respectively. They are m-dimensional vectors, which each dimension is a measure of an analysis of the source code of a module (similar to Chidamber & Kemerer (CK) metrics), and (2) y_i^S is the label, which represents the defect proneness of sample. $y_i^S \in \{-1, 1\}$, where $y_i^S = -1$ and $y_i^S = 1$ represent non-defect-prone and defect-prone modules, respectively.

3.2 Initialization stage

In the initialization stage, each sample in the source project is assigned two weights—a similarity weight Sw and a sample weight W. Sw is used to evaluate the similarity of the samples in the source and target projects, and W is used to measure the sampling probability.

Similarity weight. To measure the similarity of the samples in the source and target projects, we use a method similar to that reported by Ma et al.[17]. $\boldsymbol{X}_k^T$ ($k\in \{1, 2, \ldots, \mathrm{Tn}\}$) and $\boldsymbol{X}_i^S$ ($i \in \{1, 2, \ldots, \mathrm{Sn}\}$) are defined as follows:

$$\boldsymbol{X}_k^T = (a_{k1}^T, a_{k2}^T, \ldots, a_{kj}^T, \ldots, a_{km}^T),$$
$$\boldsymbol{X}_i^S = (a_{i1}^S, a_{i2}^S, \ldots, a_{ij}^S, \ldots, a_{im}^S).$$

Then we compute the maximum value and minimum value of the j-th attribute on $\boldsymbol{D}_T$,

$$\max_j = \max\{a_{1j}^T, a_{2j}^T, \ldots, a_{\mathrm{Tn},j}^T\} \quad (1)$$

$$\min_j = \min\{a_{1j}^T, a_{2j}^T, \ldots, a_{\mathrm{Tn},j}^T\} \quad (2)$$

S_i is the similarity value of the i-th sample in $\boldsymbol{D}_S$ and $\boldsymbol{D}_T$. S_i is defined as

$$S_i = \frac{\sum_{j=1}^{m} h(a_{ij}^S)}{m} \quad (3)$$

where $h(a_{ij}^S)$ is expressed as

$$h(a_{ij}^S) = \begin{cases} 1, & \min_j \leqslant a_{kj}^S \leqslant \max_j; \\ 0, & \text{else.} \end{cases}$$

The similarity weight of the i-th sample in $\boldsymbol{D}_S$, namely Sw_i, can be calculated as follows:

$$\mathrm{Sw}_i = \frac{S_i}{S_{\max}} \quad (4)$$

where $S_{\max} = \max\{S_1, S_2, \ldots, S_{\mathrm{Sn}}\}$. According to Eq. (4), a higher Sw_i corresponds to a higher similarity value of the i-th sample.

Sample weight. The sample weight represents the probability of the sample being taken by the sampling method for building new train dataset in the training stage. A sample with a higher sample weight means a higher probability. In the initialization stage, we assign the same weight to each sample in the source project. The value is given as

$$W_i = \frac{1}{\mathrm{Sn}} \quad (5)$$

where W_i is the i-th sample weight in $\boldsymbol{D}_S$. This weight is updated continuously during the training stage based on the basic classifier performance on the test dataset. The update strategy is described in the next section.

3.3 Training stage

In the training stage, we use a strategy similar to boosting, in which we train a series of basic classifiers by iteration and add them to a classifier pool (ClassficationPool). At each iteration, we build a basic classifier using the following steps: First, from $\boldsymbol{D}_S$, we select a number of samples with high similar weights to create a test dataset in the current iteration. Second, we select a number of $\boldsymbol{D}_S$ samples based on their sample weights to build the training dataset, and then use this training dataset to build a basic classifier. Third, we test the new basic classifier using the test dataset, and update the sample weight based on its performance. Given a satisfactory performance by the classifier, we calculate a distribution similarity distance between the training dataset and the target project, and then add the basic classifier to the ClassficationPool.

Building test dataset. Due to the impossibility of obtaining a real label for a sample from $\boldsymbol{D}_T$, the true

distribution of data in $\boldsymbol{D}_T$ cannot be determined, which will lead to the inaccurate exclusion of data with different distribution characteristics. It is difficult to determine the performance on $\boldsymbol{D}_T$ if all data in $\boldsymbol{D}_S$ are directly used to test the trained base classifier. In each iteration, we build the test dataset using the following steps. First, we assume that samples with higher similar weights have high similarity with the characteristic distribution of data in $\boldsymbol{D}_T$. On this basis, we define Similarity_threshold. Samples whose similarity weights are greater than the Similarity_threshold in the training dataset are referred to as similarity samples. We select some samples from these similar samples at a given extraction ratio (namely the Sampling_percentage). In this way, we can build a number of test datasets that are more similar to $\boldsymbol{D}_T$ and we expect the base classifiers to perform well on these test datasets. Thus, we can integrate these classifiers to obtain better results on $\boldsymbol{D}_T$.

Building a training dataset. Due to the class imbalance problem in the datasets, the base classifier will be biased toward the majority class. Thus, it is difficult to build a basic classifier that can achieve a good performance on $\boldsymbol{D}_T$. To overcome this challenge, we present Bernoulli Sampling based on Weight (BSW), whereby the data in each class of the dataset are sampled in each iteration to ensure class balance in the training set.

BSW involves two sampling processes, each of which is composed of a number of Bernoulli distribution tests. The probability of a sample being extracted in each test is determined by the sample weight. In the implementation process, we use the cumulative method[32] for sampling shown in Algorithm 1.

In the first sampling process, we use the cumulative method to extract Psize samples in the minority class of the dataset. To reduce the impact of the unbalanced class, the Psize value should not be less than the number of minority samples. In the second sampling, we extract Sn-Psize samples in $\boldsymbol{D}_S$. After the training sets are constructed, we use them to train the basic classifier.

Algorithm 1 Cumulative

Input: Sample weight arry[ ]
Output: Index of the sampled data

```
P[ ]=new double[sample weight arry.length];
Copy arry[ ] to P[ ];
for index from 1 to P.length do
    P[index]+=P[index−1];
end for
u=random double in [0,1]×P[last element];
index=0;
for index from 0 to P.length do
    if P[index] >u then
        break;
    end if
end for
return index
```

Weight update strategy. After constructing the basic classifier, we calculate the classification error of the base classifier on the test datasets and obtain the update parameter α according to the error rate ϵ. Then the weights can be updated using the following strategy: if $\alpha > 0$, the basic classifier is effective for predicting the defects in $\boldsymbol{D}_T$. Thus, a sample in $\boldsymbol{D}_S$ that has been correctly classified by the classifier can facilitate defect prediction in $\boldsymbol{D}_T$, leading to an improvement in the sample weights. Samples that are misclassified by the classifier cannot facilitate the prediction of defects in $\boldsymbol{D}_T$, so we reduce the weights of these samples. If $\alpha < 0$, the basic classifier plays only a slight role in the predicting defects in $\boldsymbol{D}_T$. Therefore, the samples in $\boldsymbol{D}_S$ that are correctly classified by the classifier will not be helpful in the prediction of defects in $\boldsymbol{D}_T$, so the weights of these samples are reduced. The samples misclassified by the classifier may facilitate defect prediction in $\boldsymbol{D}_T$, leading to improvement in the weight of these samples. Algorithm 2 shows the details of the weight update strategy.

Assignment of classifier weight. Next, we measure the similarity of each new trained dataset with that of the target project dataset. Similar to Ref. [31], we first define a feature vector $\boldsymbol{P}$ of the distribution as follows:

$$\boldsymbol{P} = (\boldsymbol{p}_1, \ldots, \boldsymbol{p}_j, \ldots, \boldsymbol{p}_m),$$

where $\boldsymbol{p}_j = (\max_j, \min_j, \text{mean}_j, \text{std}_j)$, and mean_j and std_j represent the mean and variance of all the j-dimensional feature values in the dataset, respectively. We define dis_i as the degree of similarity between the i-th training dataset (built in the i-th iteration) and $\boldsymbol{D}_T$. dis_i can be defined as follows:

$$\text{dis}_i = \frac{1}{\log(\text{ED}(\boldsymbol{P}_i, \boldsymbol{P}_T) + 1)} \tag{6}$$

where $\boldsymbol{P}_i$ is a feature vector of the distribution of the i-th train dataset, $\boldsymbol{P}_T$ is a feature vector of the distribution of $\boldsymbol{D}_T$, and ED( ) is the Euclidean distance. After all the classifiers have been constructed, the weight β_i for the i-th basic classifier is expressed as follows:

$$\beta_i = \frac{\text{dis}_i}{\sum_{i=1}^{\text{LoopNum}} \text{dis}_i} \tag{7}$$

Algorithm 2 UpdateWeight

Input: Classifier; α; $\boldsymbol{D}_S$; Similarity_threshold

1: **for** $(\boldsymbol{X}_i^S, y_i^S)$ in $\boldsymbol{D}_S$ **do**
2: $v = W_i$;
3: $y_{\text{predict}} = h(\boldsymbol{X}_i^S)$; //$h(\,)$ is the prediction function of the classifier
4: **if** $\text{Sw}_i \geqslant$ Similarity_threshold **then**
5:
$$v = \begin{cases} v \times \mathrm{e}^{\alpha \times \text{Sw}_i}, & y_{\text{predict}} = y_i^S; \\ v \times \mathrm{e}^{-\alpha \times \text{Sw}_i}, & \text{otherwise}; \end{cases}$$
6: **else**
7: **if** $\alpha > 0$ **then**
8:
$$v = \begin{cases} v \times \mathrm{e}^{\alpha \times S\text{w}_i}, & y_{\text{predict}} = y_i^S; \\ v \times \mathrm{e}^{-\alpha}, & \text{otherwise}; \end{cases}$$
9: **else**
10:
$$v = \begin{cases} v \times \mathrm{e}^{\alpha}, & y_{\text{predict}} = y_i^S; \\ v \times \mathrm{e}^{-\alpha \times \text{Sw}_i}, & \text{otherwise}; \end{cases}$$
11: **end if**
12: **end if**
13: W_i=v;
14: **end for**
15: $Z = \sum_{i=1}^{\text{Sn}} W_i$;
16: **for** $(\boldsymbol{X}_i^S, y_i^S)$ in $\boldsymbol{D}_S$ **do**
17: $W_i = W_i / Z$
18: **end for**

where LoopNum represents the number of basic classifiers.

3.4 Prediction stage

After constructing the basic classifiers, the final classifier is constructed by integrating the basic classifiers in the ClassficationPool according to their weights, which is expressed as follows:

$$y_{\text{final}} = \text{sign}\left(\sum_{i=1}^{\text{LoopNum}} \beta_i h_i(\boldsymbol{X}_k^T)\right) \tag{8}$$

where $h_i(\,)$ is the prediction function of the i-th basic classifier, y_{final} is the prediction result of ensemble classifier, $y_{\text{final}} \in \{-1, 1\}$.

3.5 Pseudo-code description

Algorithm 3 shows the pseudo-code of the TSboostDF algorithm. Lines 1–7 comprise the initialization stage. In Line 3, we calculate the dimensions of each attribute value of the target project based on the method proposed in Section 3.2. In Lines 4–7, we initialize the sample weights and similar weights for each of the data in the

Algorithm 3 TSboostDF

Input: $\boldsymbol{D}_S$=($\boldsymbol{X}_S$, y_S); $\boldsymbol{D}_T$=$\boldsymbol{X}_T$; Similarity_threshold; Sampling_percentage; LoopNum; Psize

1: /*Initialization stage*/
2: ClassficationPool=[]; //Create a classfication pool for store classifier
3: TargetInformation=CaluteTargetInformation($\boldsymbol{D}_T$);
4: **for** each d_i in $\boldsymbol{D}_S$ **do**
5: W_i =1/Sn;
6: Sw$_i$=CalcualtionSimilarWeight(d_i, TargetInformation);
7: **end for**
8: /*Training stage*/
9: i=0;
10: **while** i<LoopNum **do**
11: TestData = CreateTestData ($\boldsymbol{D}_S$, Similarity_threshold, Sampling_percentage);
12: α=–1;
13: **while** α <0 **do**
14: TrainData=BSW($\boldsymbol{D}_S$, Psize); //Built train dataset
15: Classifier=Train(TrainData);
16: ϵ=CalculationError(Classifier, TestData); //Calculate the error rate
17:
$$\alpha = 0.5 \times \log\left(\frac{1-\epsilon}{\epsilon}\right);$$
18: UpdateWeight(Classifier, α, $\boldsymbol{D}_S$, Similarity_threshold);
19: **end while**
20: dis$_i$=Dis($\boldsymbol{D}_T$, TrainData);
21: Set dis$_i$ to Classifier;
22: Add Classifier to ClassficationPool;
23: i++;
24: **end while**
25: AssignmentWeight(ClassficationPool);
26: /*Prediction stage*/
27:
$$y_{\text{final}} = \text{sign}\left(\sum_{i=1}^{\text{LoopNum}} \beta_i h_i(\boldsymbol{X}_k^T)\right).$$

$\boldsymbol{D}_S$ using Eqs. (4) and (5). Lines 8–25 comprise the training stage. In Line 11, we build the test dataset using the method described in Section 3.3. In Line 14, $\boldsymbol{D}_S$ samples are extracted based on their sample weights determined by the BSW method. Then, we can build the training datasets and basic classifiers. In Lines 16 and 17, we calculate the error rate ϵ and the error coefficient α of the basic classifier in the test datasets (note that the weight of a sample is not considered when calculating this error rate). Then, the weight is updated by the UpdateWeight algorithm. If $\alpha > 0$, the difference in the distributions of the training dataset and $\boldsymbol{D}_T$ is calculated using Eq. (6) in Line 18. If $\alpha \leqslant 0$, we rebuild the training dataset and the basic classifier. Finally, to predict the

test samples, all the classifiers are integrated with their weights in Line 27. The weights of basic classifiers are calculated using Eq. (7).

4 Experiment

In this section, we describe three experiments we conducted to verify the performance of TSboostDF. (1) In the first experiment, on different CPDP datasets, we compared the performance of TSboostDF with those of other methods. (2) To overcome class imbalance, in the second experiment we compared the performance of TSboostDF with those of other methods with respect to class imbalance. In addition we performed a statistical analysis of the results of experiments 1 and 2. (3) In this experiment, we compared the performances of TSboostDF with those of other methods for different Psize values and evaluated their effect on the performance of the classifiers.

4.1 Datasets

We used seven projects (including arc, xalan, and tomcat etc.) in the open dataset Promise[33] to build our CPDP experimental datasets. For each experiment, two projects were selected, one of which served as the target project and the other was the source project. In total, we constructed 42 CPDP experimental datasets. Each sample from the datasets represented the test results for a program module in the project. In our experiment, each sample had 20 attributes and 1 label. Each feature represented a measure of the CK metrics analysis of the source code of the module, and the label represented the defect proneness of the module. Details of those attributes are shown in Table 1[34]. In Table 1, the BUG is a label, and when its value is greater than 0, it represents a defect-prone module, otherwise represents a non-defect-prone module. These datasets can be downloaded from the website (http://openscience.us/repo/defect/). Table 2 provides details of the datasets. The first, second, third, and fourth columns in Table 2 correspond to the project name, the number of modules in the project, the number of defect-prone modules, and a description of the project, respectively. As shown in Table 2, there are far fewer defect-prone modules than non-defect-prone modules in most projects. Specifically, only 16, 27, and 77 defect-prone modules are present in sysnapse1.0, arc, and tomcat, respectively.

4.2 Performance measures

For this paper, the performance of each algorithm was determined based on its values of F1-measure, G-mean, Matthews Correlation Coefficient (MCC), and Balance, which were calculated from the Recall and Precision values defined in the obfuscation matrix in Table 3, where TP means True Positive, FP means False Positive, FN means False Negative, and TN means True Negative.

Recall. It is a measure of completeness, describing probabilities of true defective modules in comparison

Table 2 Details of experiment dataset.

Project	Number of modules	Number of defect-prone modules	Description
ant1.7	745	166	Open-source
arc	234	27	Academic
jedit3.2	272	90	Open-source
sysnapse1.0	157	16	Open-source
tomcat	858	77	Open-source
xalan2.4	723	110	Open-source
xerces1.2	440	71	Open-source

Table 1 Descriptions of sofware metrics.

Attribute	Description	Attribute	Description
WMC	Weighted methods per class	MFA	Measure of function abstraction
DIT	Depth of inheritance tree	CAM	Cohesion among methods of class
NOC	Number of children	IC	Inheritance coupling
RFC	Response for a class	AMC	Average method complexity
LCOM	Lack of cohesion in methods	CA	Afferent couplings
LOCM3	Normalized version of LCOM	CE	Efferent couplings
NPM	Number of public methods	MAX_CC	Maximum values of methods in the same class
LOC	Lines of code	AVG_CC	Mean values of methods in the same class of methods in a given class
DAM	Data access metric		
MOA	Measure of aggregation	CBM	Coupling between methods
BUG	Number of bugs detected in the number of program module	CBO	Coupling between object classes

Table 3 Obfuscation matrix.

Predicted	Actual	
	Defect	Non-defect
Defect	TP	FP
Non-defect	FN	TN

with the total number of defective modules,

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

Precision. This is a measure of exactness, which defines the probabilities of the presence of modules that are truly defective from the number of modules predicted to be defective,

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}.$$

Probability of the False alarm (PF). It shows the proportion of all the modules without defects predicted to be defective,

$$\text{PF} = \frac{\text{FP}}{\text{FP} + \text{TN}}.$$

F1-measure. It is the harmonic average of recall and precision. An algorithm with a higher F1-measure value implies a better performance,

$$\text{F1-measure} = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}}.$$

G-mean[23]. G-mean is the geometric mean of the recall values of the defective and non-defective classes,

$$\text{G-mean} = \sqrt{\text{Recall} \times (1 - \text{PF})}\,.$$

Balance[23]. Balance is introduced by calculating the Euclidean distance from the real (Recall, PF) point to (1, 0), since the point (Recall = 1, PF = 0) is the ideal point where all defects are detected without missing,

$$\text{Balance} = 1 - \frac{\sqrt{(0 - \text{PF})^2 \times (1 - \text{Recall})^2}}{\sqrt{2}}\,.$$

MCC[35]. MCC takes into account the values of TP, FP, FN, and TN more comprehensively and can be utilized when the class is imbalanced[36]. MCC values range between [–1, 1], where 1 denotes a perfect prediction and –1 indicates complete disagreement between the actual and predicted values. The higher the values, the better the performances of the classifiers.

$$\text{MCC} = \frac{\text{TP} \times \text{TN} + \text{FN} \times \text{FP}}{\sqrt{(\text{TP}+\text{FN})(\text{TP}+\text{FP})(\text{FN}+\text{TN})(\text{FP}+\text{TN})}}\,.$$

The experiments were repeated at least 10 times to obtain average value.

4.3 Statistical analysis

To analyze the experimental results and demonstrate the superiority of TSboostDF, we used the Wilcoxon rank-sum test, Cliff's delta, and box plots, which are routinely used in SDP studies[17, 35, 37, 38].

The Wilcoxon rank-sum test is a non-parametric statistical test method for determining whether the differences between two independent sample sets are statistically significant (which happens when the probability value (p-value) is less than the determined level of significance). In this study, we used the Wilcoxon rank-sum test at the 1% significance level. We utilized the measure of win/tie/lose in the Wilcoxon rank-sum test, where win/lose indicates the number of datasets for which the average value of TSboostDF's performance metrics is higher/lower than the corresponding baselines at the 1% significance level. A tie indicates the number of datasets for which the p-value is higher than the significance level.

Cliff's delta is a measure of the non-parametric effect size for quantifying the difference between two groups of samples, the value of which ranges from –1 to 1. The smaller the absolute value of the Cliff's delta measure, the more similar the values in two groups[39, 40]. Table 4 shows the mapping results of the Cliff's delta and effect size levels.

Tables 5–8 present the results of the effect size obtained by Cliff's delta, where $*$ indicates that the proposed method signifificantly outperforms the corresponding baseline with large effect sizes.

The box plot is a statistical chart used to display the distribution of a dataset, in which there are five numerical aspects: minimum, lower quartile, median, upper quartile, and maximum.

4.4 Experimental settings

To consider the performance of the TSboostDF algorithm, we compared its performances with those of the traditional machine-learning algorithm Naive Bayes (NB), the ensemble learning algorithms adaBoost[40] and adaBoost.NC[23, 41], the transfer-learning algorithm NN[29], and TNB[17]. In these CPDP experimental datasets, all the algorithms were implemented using the well-known machine-learning framework Weka[42].

Table 4 Mapping results of cliff's delta and effect size.

Cliff's delta (δ)	Effect size level
$\lvert\delta\rvert < 0.1470$	Negligible
$0.1470 \leqslant \lvert\delta\rvert < 0.3300$	Small
$0.3300 \leqslant \lvert\delta\rvert < 0.4740$	Medium
$\lvert\delta\rvert \geqslant 0.4740$	Large

Table 5 F1-measure of TSboostDF and other algorithms.

Source⇒Target	NB	TSboostDF	TNB	adaBoost	adaBoost.NC	NN
xerces1.2⇒arc	0.407 41	0.324 29	0.229 17(*)	0.407 41	0.361 57	0.435 85
tomcat⇒arc	0.216 22(*)	**0.367 31**	0.329 11(*)	0.125 00(*)	0.214 24(*)	0.273 15(*)
xalan2.4⇒arc	0.243 90(*)	**0.360 26**	0.317 07(*)	0.300 00(*)	0.264 53(*)	0.337 11(*)
sysnapse1.0⇒arc	0.372 88	0.286 87	0.333 33	0.266 67(*)	0.325 02	0.389 45
ant1.7⇒arc	0.400 00	0.315 68	0.346 67	0.400 00	0.381 88	0.358 39
jedit3.2⇒arc	0.311 11(*)	0.321 07	0.357 89	0.311 11(*)	0.285 56	0.402 02
arc⇒xerces1.2	0.210 53	**0.219 15**	0.194 29(*)	0.210 53	0.194 46(*)	0.202 64
tomcat⇒xerces1.2	0.220 34(*)	0.237 11	0.246 75	0.087 91(*)	0.223 99	0.238 33
xalan2.4⇒xerces1.2	0.226 09	0.210 94	0.232 88	0.163 27(*)	0.235 33	0.244 90
sysnapse1.0⇒xerces1.2	0.180 33(*)	0.215 57	0.218 39	0.123 71(*)	0.170 07(*)	0.181 56(*)
ant1.7⇒xerces1.2	0.240 60	0.215 63	0.265 73	0.240 60	0.240 73	0.238 09
jedit3.2⇒xerces1.2	0.206 35(*)	0.224 62	0.230 30	0.206 35(*)	0.218 27	0.225 86
arc⇒tomcat	0.358 11	**0.362 87**	0.324 93(*)	0.358 11	0.294 31(*)	0.341 99
xerces1.2⇒tomcat	0.372 55	0.345 25	0.240 57(*)	0.372 55	0.347 45	0.350 96
xalan2.4⇒tomcat	0.341 97(*)	**0.372 08**	0.324 93(*)	0.320 99(*)	0.357 64	0.34074(*)
sysnapse1.0⇒tomcat	0.312 76(*)	0.329 57	0.333 33	0.241 07(*)	0.288 99	0.323 64
ant1.7⇒tomcat	0.395 16	0.353 78	0.343 75(*)	0.395 16	0.382 93	0.391 09
jedit3.2⇒tomcat	0.389 47	0.378 01	0.317 28(*)	0.389 47	0.286 71	0.394 33
arc⇒xalan2.4	0.410 26	**0.411 08**	0.397 73	0.410 26	0.367 14	0.386 03(*)
xerces1.2⇒xalan2.4	0.346 50(*)	**0.388 06**	0.323 94(*)	0.346 50(*)	0.350 88(*)	0.352 47(*)
tomcat⇒xalan2.4	0.365 76(*)	0.391 12	0.419 75	0.257 67(*)	0.369 69(*)	0.340 37(*)
sysnapse1.0⇒xalan2.4	0.367 15(*)	0.403 68	0.418 60	0.209 88(*)	0.300 52(*)	0.345 70(*)
ant1.7⇒xalan2.4	0.364 82(*)	**0.420 45**	0.384 62(*)	0.364 82(*)	0.382 09(*)	0.384 89(*)
jedit3.2⇒xalan2.4	0.333 33(*)	**0.399 89**	0.396 91	0.333 33(*)	0.321 34(*)	0.366 21(*)
arc⇒sysnapse1.0	0.417 91(*)	**0.490 50**	0.315 79(*)	0.417 91(*)	0.339 68(*)	0.408 62(*)
xerces1.2⇒sysnapse1.0	0.300 00	0.290 24	0.244 90(*)	0.300 00	0.274 52	0.373 79
tomcat⇒sysnapse1.0	0.206 90(*)	0.402 83	0.329 11(*)	0.095 24(*)	0.256 41(*)	0.466 06
xalan2.4⇒sysnapse1.0	0.466 67	0.407 04	0.352 94(*)	0.482 76	0.451 39	0.456 64
ant1.7⇒sysnapse1.0	0.489 80	0.372 45	0.400 00	0.489 80	0.450 84	0.411 53
jedit3.2⇒sysnapse1.0	0.266 67(*)	0.417 19	0.307 69(*)	0.266 67(*)	0.286 53(*)	0.433 12
arc⇒ant1.7	0.486 67(*)	**0.536 60**	0.511 06(*)	0.486 67(*)	0.460 23	0.466 69(*)
xerces1.2⇒ant1.7	0.468 09(*)	**0.539 29**	0.476 19(*)	0.468 09(*)	0.481 04(*)	0.453 31(*)
tomcat⇒ant1.7	0.378 60(*)	0.562 32	0.572 13	0.135 42(*)	0.410 17(*)	0.406 28(*)
xalan2.4⇒ant1.7	0.408 33(*)	0.534 89	0.509 71(*)	0.382 61(*)	0.449 07(*)	0.409 19(*)
sysnapse1.0⇒ant1.7	0.351 93(*)	0.521 72	0.537 63	0.189 05(*)	0.370 35(*)	0.346 73(*)
jedit3.2⇒ant1.7	0.475 00(*)	**0.551 28**	0.536 59(*)	0.475 00(*)	0.461 06(*)	0.506 50(*)
arc⇒jedit3.2	0.361 29(*)	0.412 19	0.568 05	0.361 29(*)	0.373 37	0.354 20
xerces1.2⇒jedit3.2	0.406 02(*)	**0.578 43**	0.566 52(*)	0.406 02(*)	0.462 19(*)	0.435 83(*)
tomcat⇒jedit3.2	0.393 94(*)	**0.592 96**	0.566 04(*)	0.166 67(*)	0.429 72(*)	0.473 08(*)
xalan2.4⇒jedit3.2	0.336 00(*)	0.475 69	0.551 72	0.302 52(*)	0.406 51(*)	0.331 88(*)
sysnapse1.0⇒jedit3.2	0.308 94(*)	0.477 91	0.553 46	0.130 84(*)	0.302 46(*)	0.335 81(*)
ant1.7⇒jedit3.2	0.584 42(*)	**0.639 77**	0.573 25(*)	0.584 42(*)	0.595 33(*)	0.524 18(*)
Average	0.350 02	**0.396 61**	0.380 97	0.309 13	0.343 48	0.367 60

Details regarding these algorithms are provided below.

adaBoost. adaBoost is an ensemble learning algorithm consisting of basic classifiers with weights. The number of iterations is 20, and the basic classifier is NB.

adaBoost.NC. adaBoost.NC is an extension to adaBoost that considers the class imbalance in datasets. The number of iterations is 20 and the basic classifier is NB, as reported in Ref. [23]. As a prerequisite, random oversampling was first applied to the minority class to ensure that the classes were of the same size.

TNB. This is a transfer-learning algorithm based on sample weights. In this paper, we used Weka's unsupervised discretization method to discretize the data

Table 6 MCC obtained by TSboostDF and other algorithms.

Source⇒Target	NB	TSboostDF	TNB	adaBoost	adaBoost.NC	NN
xerces1.2⇒arc	0.330 11	**0.219 36**	0.089 13(*)	0.330 11	0.266 32	0.362 66
tomcat⇒arc	0.188 23(*)	**0.273 78**	0.225 22(*)	0.131 64(*)	0.152 02(*)	0.190 47(*)
xalan2.4⇒arc	0.190 89(*)	**0.263 93**	0.209 90(*)	0.262 78	0.199 38(*)	0.259 64
sysnapse1.0⇒arc	0.284 50	0.169 98	0.230 53	0.196 93	0.228 40	0.306 41
ant1.7⇒arc	0.320 19	0.208 16	0.247 17	0.320 19	0.295 01	0.265 69
jedit3.2⇒arc	0.247 13	0.215 34	0.269 68	0.247 13	0.178 61	0.323 07
arc⇒xerces1.2	0.046 70	**0.058 64**	0.003 17(*)	0.046 70	0.011 09(*)	0.042 44
tomcat⇒xerces1.2	0.108 33	0.079 06	0.088 54	0.022 92(*)	0.097 83	0.117 82
xalan2.4⇒xerces1.2	0.121 50	0.024 02	0.080 47	0.093 79	0.102 37	0.130 12
sysnapse1.0⇒xerces1.2	0.053 47	0.046 15	0.034 72(*)	0.047 28	0.039 44	0.056 17
ant1.7⇒xerces1.2	0.106 46	0.017 67	0.123 28	0.106 46	0.093 32	0.099 30
jedit3.2⇒xerces1.2	0.077 06	0.040 44	0.057 76	0.077 06	0.011 82	0.085 90
arc⇒tomcat	0.311 88	**0.315 78**	0.285 89(*)	0.311 88	0.235 31(*)	0.286 44
xerces1.2⇒tomcat	0.305 48	**0.306 81**	0.165 01(*)	0.305 48	0.284 02	0.280 73
xalan2.4⇒tomcat	0.269 40(*)	**0.329 58**	0.285 89(*)	0.250 77(*)	0.294 95	0.268 11(*)
sysnapse1.0⇒tomcat	0.238 49(*)	0.281 68	0.286 70	0.149 43(*)	0.217 68	0.252 85(*)
ant1.7⇒tomcat	0.343 54	0.319 73	0.300 41(*)	0.343 54	0.337 04	0.340 88
jedit3.2⇒tomcat	0.323 88(*)	**0.338 64**	0.272 64(*)	0.323 88(*)	0.221 95(*)	0.331 78
arc⇒xalan2.4	0.287 53	**0.292 16**	0.270 78(*)	0.287 53	0.246 34	0.259 85(*)
xerces1.2⇒xalan2.4	0.198 46(*)	**0.265 06**	0.162 44(*)	0.198 46(*)	0.203 31(*)	0.206 45(*)
tomcat⇒xalan2.4	0.235 72(*)	0.260 40	0.298 99	0.191 14(*)	0.234 52(*)	0.209 39(*)
sysnapse1.0⇒xalan2.4	0.262 61	0.283 90	0.296 45	0.135 47(*)	0.200 57(*)	0.244 36
ant1.7⇒xalan2.4	0.225 12(*)	**0.312 55**	0.251 19(*)	0.225 12(*)	0.248 75(*)	0.252 04(*)
jedit3⇒xalan2.4	0.177 57(*)	**0.281 00**	0.274 72	0.177 57(*)	0.161 22(*)	0.225 76(*)
arc⇒sysnapse1.0	0.395 73(*)	**0.465 99**	0.231 10(*)	0.395 73(*)	0.295 83(*)	0.379 42(*)
xerces1.2⇒sysnapse1.0	0.207 94(*)	0.237 57	0.153 56(*)	0.207 94(*)	0.168 24	0.334 08
tomcat⇒sysnapse1.0	0.127 98(*)	0.355 21	0.282 62(*)	0.058 81(*)	0.174 79(*)	0.420 96
xalan2.4⇒sysnapse1.0	0.411 72	0.349 88	0.299 74(*)	0.433 57	0.388 93	0.399 28
ant1.7⇒sysnapse1.0	0.446 30	0.333 25	0.342 32	0.446 30	0.409 71	0.376 15
jedit3.2⇒sysnapse1.0	0.190 10(*)	0.370 71	0.244 70(*)	0.190 10(*)	0.201 79(*)	0.380 70
arc⇒ant1.7	0.362 32(*)	**0.419 40**	0.346 83(*)	0.362 32 (*)	0.323 04	0.343 32(*)
xerces1.2⇒ant1.7	0.357 21(*)	**0.385 80**	0.289 44(*)	0.357 21(*)	0.329 81(*)	0.337 99(*)
tomcat⇒ant1.7	0.305 61(*)	0.427 35	0.432 46	0.126 66(*)	0.315 00(*)	0.320 83(*)
xalan2.4⇒ant1.7	0.350 61(*)	**0.396 13**	0.344 22(*)	0.342 32(*)	0.370 26	0.352 14(*)
sysnapse1.0⇒ant1.7	0.293 95(*)	0.383 41	0.390 15	0.170 75(*)	0.282 77(*)	0.283 15(*)
jedit3.2⇒ant1.7	0.332 05(*)	**0.404 65**	0.381 60(*)	0.332 05(*)	0.276 50(*)	0.362 66(*)
arc⇒jedit3.2	0.118 96(*)	0.239 64	0.376 25	0.118 96(*)	0.153 77	0.123 58(*)
xerces1.2⇒jedit3.2	0.273 54(*)	**0.360 97**	0.292 36(*)	0.273 54(*)	0.276 23(*)	0.300 30(*)
tomcat⇒jedit3.2	0.261 71(*)	**0.416 89**	0.398 10	0.095 68(*)	0.280 00(*)	0.340 81(*)
xalan2.4⇒jedit3.2	0.219 79(*)	0.286 83	0.341 72	0.212 77(*)	0.259 05	0.204 51(*)
sysnapse1.0⇒jedit3.2	0.193 38(*)	0.307 24	0.380 14	0.044 38(*)	0.183 61(*)	0.221 86(*)
ant1.7⇒jedit3.2	0.438 83(*)	**0.467 51**	0.414 02(*)	0.438 83(*)	0.430 54(*)	0.364 40(*)
Average	0.251 00	**0.281 24**	0.256 00	0.223 60	0.230 50	0.267 73

for TNB.

NN. This is a transfer-learning algorithm based on K-nearest neighbors. In our experiment, the number of K-nearest neighbors is 10, and NB is used as the basic classifier.

TSboostDF. The number of iterations of this algorithm is 20, and the basic classifier is NB with a Similarity_threshold value of 0.75, Sampling_percentage of 0.8, and Psize of $0.5 \times$ Sn.

4.5 Experiment on different CPDP datasets

In this experiment, we calculated F1-measure and MCC by each of the above methods to determine their overall performance in CPDP. The experimental

Table 7 G-mean obtained by TSboostDF and other algorithms.

Source⇒Target	NB	TSboostDF	TNB	adaBoost	adaBoost.NC	NN
xerces1.2⇒arc	0.613 12	0.611 82	0.541 53(*)	0.613 12	0.618 87	0.635 87
tomcat⇒arc	0.379 28(*)	0.622 70	0.625 11	0.270 19(*)	0.396 55(*)	0.478 83(*)
xalan2.4⇒arc	0.420 87(*)	**0.637 63**	0.619 51(*)	0.463 37(*)	0.464 14(*)	0.539 96(*)
sysnapse1.0⇒arc	0.605 04	0.594 67	0.626 97	0.457 54(*)	0.577 01	0.608 92
ant1.7⇒arc	0.611 51(*)	0.620 13	0.632 51	0.611 51(*)	0.616 40	0.601 12(*)
jedit3.2⇒arc	0.495 46(*)	0.614 75	0.688 84	0.495 46(*)	0.554 89(*)	0.611 83
arc⇒xerces1.2	0.430 88	**0.437 97**	0.427 77	0.430 88	0.423 29	0.418 84(*)
tomcat⇒xerces1.2	0.407 71(*)	0.458 58	0.470 31	0.232 15(*)	0.419 91(*)	0.432 11(*)
xalan2.4⇒xerces1.2	0.409 53(*)	0.447 06	0.449 22	0.326 92(*)	0.436 30	0.435 29(*)
sysnapse1.0⇒xerces1.2	0.371 67(*)	0.441 03	0.454 63	0.282 71(*)	0.363 98(*)	0.371 00(*)
ant1.7⇒xerces1.2	0.444 14(*)	0.459 86	0.478 72	0.444 14(*)	0.454 58	0.444 83(*)
jedit3.2⇒xerces1.2	0.402 81(*)	**0.462 54**	0.461 75	0.402 81(*)	0.438 32	0.431 98(*)
arc⇒tomcat	0.736 21	0.731 91	0.734 26	0.736 21	0.670 84	0.707 73
xerces1.2⇒tomcat	0.661 26(*)	**0.744 62**	0.641 34(*)	0.661 26(*)	0.686 93(*)	0.631 97(*)
xalan2.4⇒tomcat	0.618 89(*)	**0.748 42**	0.734 26(*)	0.558 71(*)	0.669 93(*)	0.612 71(*)
sysnapse1.0⇒tomcat	0.642 36(*)	0.721 83	0.726 63	0.544 77 (*)	0.611 34(*)	0.655 66(*)
ant1.7⇒tomcat	0.732 77(*)	**0.755 52**	0.736 44(*)	0.732 77(*)	0.744 21(*)	0.735 90(*)
jedit3.2⇒tomcat	0.658 60(*)	**0.756 55**	0.722 78(*)	0.658 60(*)	0.643 12(*)	0.683 89(*)
arc⇒xalan2.4	0.648 25	0.645 83	0.676 62	0.648 25	0.611 45	0.619 75
xerces1.2⇒xalan2.4	0.617 45(*)	**0.680 13**	0.611 98(*)	0.617 45(*)	0.634 48(*)	0.624 84(*)
tomcat⇒xalan2.4	0.597 97(*)	0.665 27	0.686 26	0.425 37(*)	0.620 69(*)	0.566 97(*)
sysnapse1.0⇒xalan2.4	0.558 75(*)	0.632 43	0.673 15	0.381 73(*)	0.496 23(*)	0.533 64(*)
ant1.7⇒xalan2.4	0.626 09(*)	**0.711 13**	0.647 38(*)	0.626 09(*)	0.653 77(*)	0.643 24(*)
jedit3.2⇒xalan2.4	0.614 93(*)	**0.690 23**	0.685 91	0.614 93(*)	0.574 39(*)	0.647 91(*)
arc⇒sysnapse1.0	0.803 36	**0.832 15**	0.659 42(*)	0.803 36	0.712 04(*)	0.789 91(*)
xerces1.2⇒sysnapse1.0	0.571 95(*)	0.678 62	0.614 54(*)	0.571 95(*)	0.595 65(*)	0.758 75
tomcat⇒sysnapse1.0	0.417 38(*)	0.762 43	0.724 14(*)	0.246 43(*)	0.480 47(*)	0.788 14
xalan2.4⇒sysnapse1.0	0.644 81(*)	0.745 21	0.732 96(*)	0.647 21(*)	0.686 80(*)	0.751 15
ant1.7⇒sysnapse1.0	0.798 94	0.759 42	0.742 28(*)	0.798 94	0.788 78	0.784 48
jedit3.2⇒sysnapse1.0	0.481 94(*)	**0.769 02**	0.695 73(*)	0.481 94(*)	0.576 64(*)	0.762 34
arc⇒ant1.7	0.627 24(*)	0.670 47	0.691 57	0.627 24(*)	0.616 04	0.608 68(*)
xerces1.2⇒ant1.7	0.602 71(*)	**0.721 74**	0.671 62(*)	0.602 71(*)	0.641 87(*)	0.592 36(*)
tomcat⇒ant1.7	0.512 12(*)	0.713 97	0.742 59	0.276 69(*)	0.546 48(*)	0.539 42(*)
xalan2.4⇒ant1.7	0.531 45(*)	0.686 74	0.691 73	0.505 87(*)	0.577 28(*)	0.531 95(*)
sysnapse1.0⇒ant1.7	0.485 69(*)	0.673 90	0.701 52	0.333 61(*)	0.512 35(*)	0.482 66(*)
jedit3.2⇒ant1.7	0.629 41(*)	0.721 00	0.722 81	0.629 41(*)	0.624 42(*)	0.661 48
arc⇒jedit3.2	0.497 86	0.533 40	0.665 20	0.497 86	0.500 44	0.490 77
xerces1.2⇒jedit3.2	0.523 09(*)	**0.677 13**	0.650 44(*)	0.523 09(*)	0.575 70(*)	0.547 45(*)
tomcat⇒jedit3.2	0.513 31(*)	**0.683 50**	0.658 84(*)	0.308 31(*)	0.544 53(*)	0.576 69(*)
xalan2.4⇒jedit3.2	0.464 10(*)	0.587 72	0.654 09	0.433 49(*)	0.522 23(*)	0.461 87(*)
sysnapse1.0⇒jedit3.2	0.441 44(*)	0.586 88	0.649 41	0.271 12(*)	0.432 11(*)	0.463 27(*)
ant1.7⇒jedit3.2	0.669 18(*)	**0.723 62**	0.662 99(*)	0.669 18(*)	0.682 99(*)	0.622 66(*)
Average	0.56004	**0.652 85**	0.644 90	0.511 08	0.571 39	0.592 59

results are shown in Tables 5 and 6, in which each CPDP experimental dataset is described by the format (source project ⇒ target project). We can see that TSboostDF obtained the highest F1-measure value on 16 datasets and the highest MCC value on 20 datasets in 42 CPDP experimental datasets. Generally, it exhibited significant superiority over other algorithms on xerces1.2⇒xalan2.4, tomcat⇒arc, xerces1.2⇒ant1.7, and ant1.7⇒jedit3.2 datasets. Moreover, the average F1-measure and MCC values of TSboostDF are higher than those of other algorithms. The average F1-measure and MCC values of the traditional machine-learning algorithms, such as NB or adaBoost, are lower than that of the transfer-learning algorithm.

Table 8 Balance obtained by TSboostDF and other algorithms.

Source⇒Target	NB	TSboostDF	TNB	adaBoost	adaBoost.NC	NN
xerces1.2⇒arc	0.577 42(*)	0.594 69	0.536 49(*)	0.577 42(*)	0.594 47	0.598 37
tomcat⇒arc	0.397 30(*)	0.596 89	0.609 90	0.345 19(*)	0.409 62(*)	0.465 79(*)
xalan2.4⇒arc	0.423 02(*)	**0.617 18**	0.606 28(*)	0.449 51(*)	0.460 23(*)	0.512 71(*)
sysnapse1.0⇒arc	0.574 88	0.584 09	0.611 05	0.448 50 (*)	0.557 00	0.576 10
ant1.7⇒arc	0.576 97(*)	0.607 53	0.614 35	0.576 97(*)	0.586 60(*)	0.573 41(*)
jedit3.2⇒arc	0.474 87(*)	0.599 30	0.685 46	0.474 87(*)	0.540 86(*)	0.577 01(*)
arc⇒xerces1.2	0.438 26	**0.443 18**	0.436 95	0.438 26	0.434 55	0.429 72(*)
tomcat⇒xerces1.2	0.418 70(*)	0.458 54	0.467 80	0.332 03(*)	0.427 65(*)	0.435 47(*)
xalan2.4⇒xerces1.2	0.419 32(*)	**0.451 83**	0.450 84	0.371 51(*)	0.439 57(*)	0.437 08(*)
sysnapse1.0⇒xerces1.2	0.397 55(*)	0.446 26	0.457 68	0.351 52(*)	0.396 33(*)	0.398 58(*)
ant1.7⇒xerces1.2	0.445 19(*)	0.462 63	0.472 25	0.445 19(*)	0.454 61	0.446 20(*)
jedit3.2⇒xerces1.2	0.416 78(*)	**0.464 03**	0.462 55	0.416 78(*)	0.445 79	0.437 28(*)
arc⇒tomcat	0.733 24	0.724 70	0.733 84	0.733 24	0.668 23	0.700 26
xerces1.2⇒tomcat	0.632 90(*)	**0.744 26**	0.641 07(*)	0.632 90(*)	0.672 35(*)	0.603 77(*)
xalan2.4⇒tomcat	0.589 01(*)	**0.745 76**	0.733 84(*)	0.528 62(*)	0.645 56(*)	0.582 29(*)
sysnapse1.0⇒tomcat	0.623 57(*)	0.720 00	0.725 87	0.528 16(*)	0.599 98(*)	0.637 85(*)
ant1.7⇒tomcat	0.720 15(*)	**0.755 42**	0.735 82(*)	0.720 15(*)	0.738 11(*)	0.725 33(*)
jedit3.2⇒tomcat	0.626 28(*)	**0.754 66**	0.722 76(*)	0.626 28(*)	0.625 92(*)	0.656 96(*)
arc⇒xalan2.4	0.631 58	0.628 14	0.675 22	0.631 58	0.606 71	0.601 37
xerces1.2⇒xalan2.4	0.611 42(*)	**0.678 39**	0.611 87(*)	0.611 42(*)	0.633 16(*)	0.619 50(*)
tomcat⇒xalan2.4	0.578 91(*)	0.661 95	0.681 79	0.426 70(*)	0.607 82(*)	0.548 31(*)
sysnapse1.0⇒xalan2.4	0.532 19(*)	0.613 35	0.663 72	0.400 81(*)	0.491 41(*)	0.513 54(*)
ant1.7⇒xalan2.4	0.616 66(*)	**0.709 18**	0.639 26(*)	0.616 66(*)	0.648 84(*)	0.633 37(*)
jedit3.2⇒xalan2.4	0.612 95(*)	**0.689 11**	0.685 74	0.612 95(*)	0.566 31(*)	0.646 38(*)
arc⇒sysnapse1.0	0.794 47(*)	**0.829 12**	0.651 49(*)	0.794 47(*)	0.697 29(*)	0.783 47(*)
xerces1.2⇒sysnapse1.0	0.548 93(*)	0.660 31	0.606 96(*)	0.548 93(*)	0.585 06(*)	0.754 89
tomcat?sysnapse1.0	0.42329(*)	0.762 04	0.716 36(*)	0.336 78(*)	0.474 88(*)	0.784 31
xalan2.4⇒sysnapse1.0	0.600 71(*)	**0.740 48**	0.732 62(*)	0.601 12(*)	0.652 35(*)	0.737 06
ant1.7⇒sysnapse1.0	0.794 23	0.755 67	0.738 19(*)	0.794 23	0.785 68	0.783 19
jedit3.2⇒sysnapse1.0	0.467 30(*)	**0.767 54**	0.693 20(*)	0.467 30(*)	0.556 20(*)	0.757 11
arc⇒ant1.7	0.596 91(*)	0.641 46	0.687 36	0.596 91(*)	0.605 84	0.578 46(*)
xerces1.2⇒ant1.7	0.569 68(*)	**0.720 44**	0.671 29(*)	0.569 68(*)	0.621 94(*)	0.561 15(*)
tomcat⇒ant1.7	0.487 44(*)	0.699 04	0.740 68	0.348 08(*)	0.517 75(*)	0.510 67(*)
xalan2.4⇒ant1.7	0.500 68(*)	0.667 43	0.688 28	0.479 75(*)	0.543 83(*)	0.501 12(*)
sysnapse1.0⇒ant1.7	0.466 59(*)	0.653 42	0.690 49	0.373 52(*)	0.498 72(*)	0.466 38(*)
jedit3.2⇒ant1.7	0.604 97(*)	0.715 68	0.722 77	0.604 97(*)	0.607 79(*)	0.640 35(*)
arc⇒jedit3.2	0.492 11	0.516 44	0.648 72	0.492 11	0.510 99	0.486 09
xerces1.2⇒jedit3.2	0.501 14(*)	**0.670 76**	0.646 37(*)	0.501 14(*)	0.556 38(*)	0.521 82(*)
tomcat⇒jedit3.2	0.493 34(*)	**0.664 81**	0.634 36(*)	0.362 64(*)	0.521 94(*)	0.546 42(*)
xalan2.4⇒jedit3.2	0.455 16(*)	0.566 81	0.641 60	0.432 70(*)	0.504 30(*)	0.454 54(*)
sysnapse1.0⇒jedit3.2	0.439 53(*)	0.563 08	0.625 77	0.346 73(*)	0.442 38(*)	0.455 94(*)
ant1.7⇒jedit3.2	0.638 82(*)	**0.711 51**	0.636 26(*)	0.638 82(*)	0.660 79(*)	0.594 75(*)
Average	0.54630	**0.644 22**	0.638 93	0.513 98	0.561 81	0.577 96

Although the traditional machine-learning algorithm has difficulty obtaining good performance due to the different distributions of the datasets in different projects, information from the source project can be used by the transfer-learning algorithm to enhance model training, which can reduce the influence of the different dataset distributions of different projects. Therefore, the transfer-learning algorithm is more suitable for CPDP. On the other hand, although the average F1-measure and MCC values of adaBoost.NC are higher than those of adaBoost, the impact of the distribution differences in CPDP datasets is not considered, which leads to difficulties in achieving better performance. In addition, because the imbalance problem in datasets is not considered by TNB

and NN algorithms, the performances of these algorithms are inferior to that of the TSboostDF algorithm.

4.6 Experiments for overcoming class imbalance

In this section, we calculate G-mean and balance of each algorithm on all datasets, which we use to compare their performances on datasets with class imbalance problems[23]. The experimental results, as shown in Tables 7 and 8, reveal that TSboostDF obtained the highest G-mean value in 16 datasets and the highest balance value in 18 datasets of the 42 CPDP experimental datasets, and the average G-mean and balance of TSboostDF are also higher than those of the other algorithms. The G-mean and balance values of adaBoost.NC are higher than those of adaBoost, which means that it can overcome the impact of class imbalance more effectively than adaBoost. However, because of the difference in the data distributions of the source and target projects, the G-mean and balance values of adaBoost.NC are lower than those of TNB and NN. TSboostDF considers not only the difference in the data distributions of the source and target projects, but also the impact of class imbalance on the dataset. Therefore, the values of G-mean and balance of TSboostDF are not only higher than those of the traditional machine-learning algorithms, but also higher than those of TNB and NN. This means that TSboostDF is more effective in overcoming the impact of the class imbalance problem.

4.7 Statistical analysis

Tables 5–9 show the results of the Cliff's delta effect size test and the Wilcoxon rank-sum test, in which we can see that compared with other methods, TSboostDF obtained a higher win than lose value in four performance metrics (although TSboostDF obtained the same win and lose values for the balance indicator, as compared with TNB) and effect size levels are large in these win datasets. Based on the above results, the performance of TSboostDF is considered to be better than those of other methods, and this result is statistically significant. In addition, from Table 10 and Figs. 1–4, we can see that the performance of TSboostDF is better than those of the other methods, and that its median values for the four performance metrics are higher than those of the other algorithms.

Table 9 Results win/tie/lose of Wilcoxon rank-sum test.

Method	F1-measure	MCC	G-mean	Balance
TSboostDF vs. NB	28/3/11	25/5/12	34/7/1	36/5/1
TSboostDF vs. TNB	23/7/12	24/8/10	18/11/13	18/6/18
TSboostDF vs. adaBoost	30/3/9	26/6/10	35/6/1	37/4/1
TSboostDF vs. adaBoost.NC	23/13/6	21/15/6	30/11/1	32/9/1
TSboostDF vs. NN	22/9/11	21/7/14	30/7/5	32/7/3

4.8 Experiment with different Psize values

This experiment was designed to evaluate the effect of Psize on the performance of the classifiers. We tested the performance of TSboostDF on different Psize values, including Opsize, 0.35Sn, 0.4Sn, 0.45Sn, and 0.5Sn, where Opsize is the number of the minority samples in the source project. The same experimental datasets were used in this experiment as those used in Section 4.5. The experimental results, as shown in Table 11 and

Table 10 Median values of each performance metric.

Method	F1-measure	MCC	G-mean	Balance
NB	0.363 05	0.262 150	0.584 961	0.559 305
TSboostDF	**0.389 59**	**0.299 490**	**0.677 870**	**0.663 380**
TNB	0.345 21	0.278 667	0.664 096	0.650 108 9
adaBoost	0.316 04	0.210 350	0.514 481	0.496 625
adaBoost.NC	0.349 16	0.234 916	0.577 140	0.561 654
NN	0.370 00	0.281 936	0.608 802	0.576 554

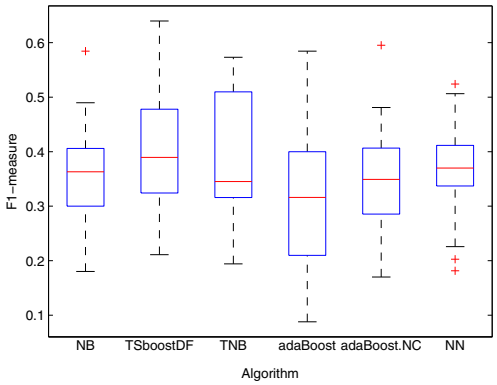


Fig. 1 Box plot of F1-measure of each algorithm.

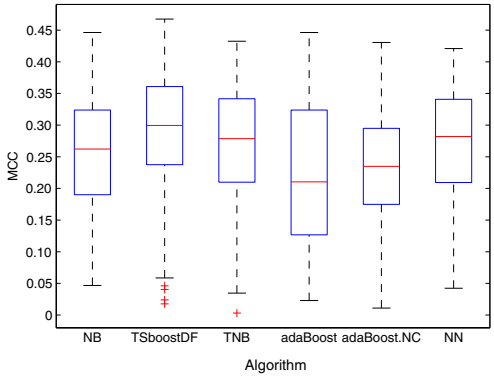


Fig. 2 Box plot of MCC of each algorithm.

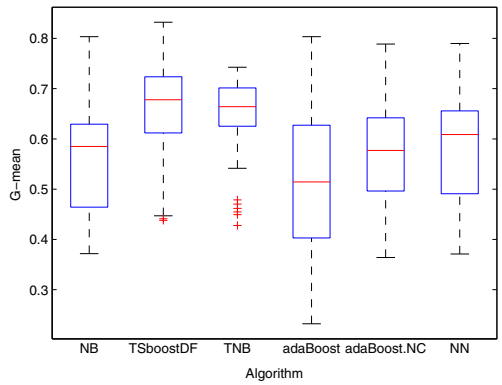


Fig. 3 Box plot of the G-mean of each algorithm.

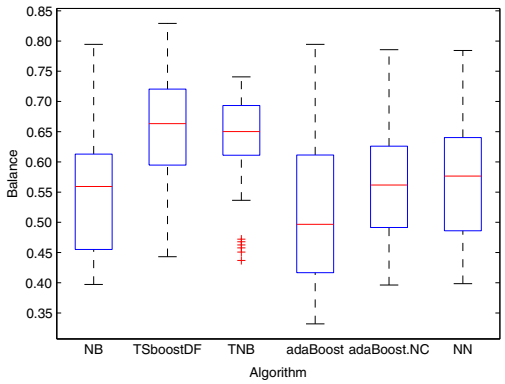


Fig. 4 Box plot of the balance of each algorithm.

Table 11 Performance of TSboostDF for different Psize values.

Psize	F1-measure	MCC	G-mean	Balance
Opsize	0.396 43	**0.285 68**	0.642 57	0.632 21
0.35Sn	0.397 31	0.283 58	0.648 39	0.638 81
0.4Sn	**0.397 61**	0.283 52	0.650 96	0.641 73
0.45Sn	0.397 40	0.282 16	0.651 95	0.643 01
0.5Sn	0.396 61	0.281 24	**0.652 85**	**0.644 22**
Best baseline	0.380 97	0.267 73	0.644 90	0.638 93

Figs. 5–8, reveal that G-mean and balance of TSboostDF increase with the increase in Psize, which shows that our sampling strategy is effective in overcoming the impact of class imbalance. However, when Psize is too large, F1-measure and MCC will be affected. As shown in Table 11, when Psize is in the range of 0.4Sn to 0.5Sn, all the performance measures of TSboostDF are higher than the best baselines, so the results are more reasonable when the Psize value is in the range of 0.4Sn to 0.5Sn.

5 Threats to Validity

5.1 Internal validity

It is similar to Refs. [35] and [39], the threats to the internal validity of our study mainly relate to the

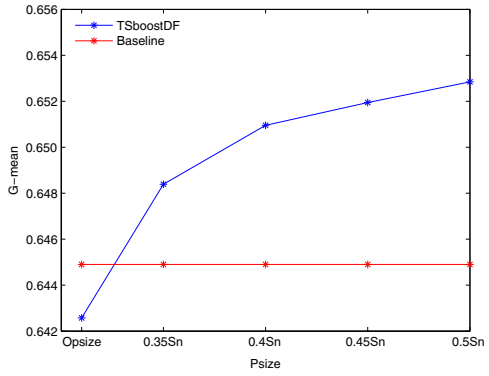


Fig. 5 G-mean of TSboostDF for different Psize values.

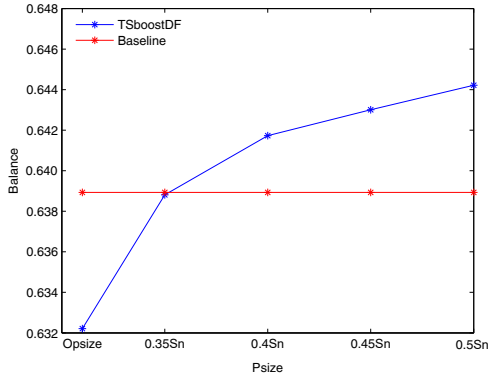


Fig. 6 Balance of TSboostDF for different Psize values.

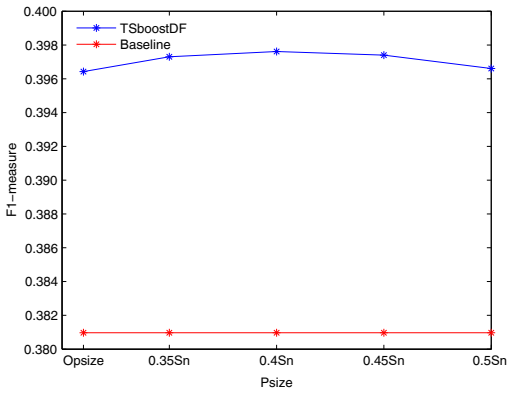


Fig. 7 F1-measure of TSboostDF for different Psize values.

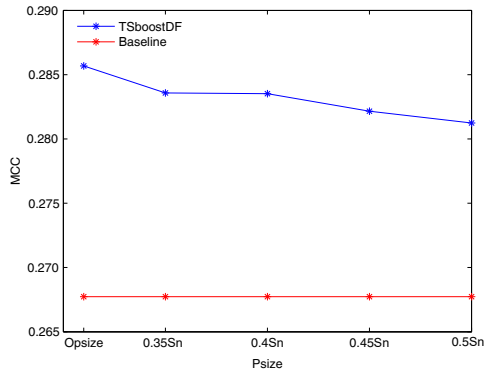


Fig. 8 MCC of TSboostDF for different Psize values.

publicly available, we carefully implemented other baselines with reference to their original studies. Although we checked our source codes carefully, errors may remain that we failed to notice.

5.2 External validity

External validity indicates the degree to generalize the research results to other situations[39]. In this study, we compared it with five SDP methods on seven public projects and use four well-known performance External validity indicates the degree to generalize the research results to other situations[39]. In this study, we compared it with five SDP methods on seven public projects and used four well-known performance measures (i.e., F1-measure, G-mean, balance, and MCC) to evaluate the performance. However, we still cannot claim that the findings will be completely suitable for other defect datasets. More defect datasets and the baseline methods should be applied to reduce this threat. In addition, we evaluated the prediction performance in terms of four well-known performance measures (i.e., F1-measure, G-mean, balance, and MCC) to ensure experimental generality.

5.3 Statistical validity

In the statistical experiment, we use Wilcoxon rank-sum test, Cliff's delta, and box plots to verify the statistical validity of the experimental results, and these methods do not rest on any assumption concerning the underlying distributions. Therefore, our experimental results are statistically significant.

6 Conclusion and Future Work

In this paper, we proposed the transfer-learning algorithm TSboostDF, which can be used to resolve the CPDP problem. TSboostDF integrates the BLS sampling method, which is based on the weight of the sample, with the transfer-learning method to overcome the shortcomings of the traditional algorithms used in CPDP. The proposed algorithm demonstrated better performance than other CPDP algorithms based on transfer-learning. The effects of multi-source transfer-learning on CPDP warrant future investigation by the integration of information from multiple source projects for knowledge transfer. This strategy will help classifiers to achieve better performance on CPDP problems.

Acknowledgment

This work was partially supported by the Army Weapons and Equipment Internal Research (No. LJ20191C080690).

References

[1] X. Chen, Q. Gu, W. S. Liu, S. L. Liu, and C. Ni, Survey of static software defect prediction, (in Chinese), *Journal of Software*, vol. 27, no. 1, pp. 1–25, 2016.

[2] Q. Wang, S. J. Wu, and M. S. Li, Software defect prediction, (in Chinese), *Journal of Software*, vol. 19, no. 7, pp. 1565–1580, 2008.

[3] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, A systematic literature review on fault prediction performance in software engineering, *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, 2012.

[4] K. Punitha and S. Chitra, Software defect prediction using software metrics—A survey, in *Proc. of International Conference on Information Communication & Embedded Systems*, Chennai, India, 2013, pp. 555–558.

[5] R. Malhotra, An empirical framework for defect prediction using machine learning techniques with Android software, *Applied Soft Computing*, vol. 49, pp. 1034–1050, 2016.

[6] R. Rana, M. Staron, C. Berger, J. Hansson, M. Nisslon, and W. Meding, The adoption of machine learning techniques for software defect prediction: An initial industrial validation, in *Proc. on Knowledge-Based Software Engineering*, Cham, Greece, 2014, pp. 270–285.

[7] M. Shepperd, D. Bowes, and T. Hall, Researcher bias: The use of machine learning in software defect prediction, *IEEE Transactions on Software Engineering*, vol. 42, no. 40, pp. 603–616, 2014.

[8] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 649–660, 2008.

[9] K. O. Elish and M. O. Elish, Predicting defect-prone software modules using support vector machines, *Journal of Systems & Software*, vol. 81, no. 5, pp. 649–660, 2008.

[10] X. Y. Jing, S. Ying, Z. W. Zhang, S. S. Wu, and J. Liu, Dictionary learning-based software defect prediction, in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, 2014, pp. 414–423.

[11] J. Wang, B. Shen, and Y. Chen, Compressed C4.5 models for software defect prediction, in *Proceedings of International Conference on Quality Software*, Nanjing, China, pp. 13–16, 2012.

[12] H. Y. Jiang, M. Zong, and X. Y. Liu, Research of software defect prediction model based on ACO-SVM, (in Chinese), *Chinese Journal of Computers*, vol. 34, no. 6, pp. 1148–1154, 2011.

[13] M. Li, H. Zhang, R. Wu, and Z. H. Zhou, Sample-based software defect prediction with active and semi-supervised learning, *Automated Software Engineering*, vol. 19, no. 2, pp. 201–230, 2012.

[14] H. Lu, E. Kocaguneli, and B. Cukic, Defect prediction between software versions with active learning and dimensionality reduction, in *Proceedings of International Symposium on Software Reliability Engineering*, Naples, Italy, 2014, pp. 270–285.

[15] L. Pelayo and S. Dick, Applying novel resampling strategies to software defect prediction, in *Proceedings of Annual Meeting of the North American Fuzzy Information Processing Society*, San Diego, CA, USA, 2007, pp. 69–72.

[16] T. Menzies, T. Burak, B. Ayse, G. Gregory, C. Bojan, and Y. Jiang, Implications of ceiling effects in defect predictors, in *Proc. of International Workshop on Predictor MODELS in Software Engineering*, Leipzig, Germany, pp. 47–54, 2008.

[17] Y. Ma, G. C. Luo, X. Zeng, and A. G. Chen, Transfer learning for cross-company software defect prediction, *Information and Software Technology*, vol. 54, no. 3, pp. 248–256, 2012

[18] S. J. Pan and Q. Yang, A survey on transfer learning, *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[19] F. Z. Zhuang, P. Luo, Q. He, and Z. Z. Shi, Survey on transfer learning research, (in Chinese), *Journal of Software*, vol. 26, no. 1, pp. 26–39, 2015.

[20] J. Nam, S. J. Pan, and S. Kim, Transfer defect learning, in *Proc. of 2013 35th International Conference on Software Engineering* (*ICSE*), San Francisco, CA, USA, 2013, pp. 382–391.

[21] L. Chen, B. Fang, Z. W. Shang, and Y. Y. Tang, Negative samples reduction in cross-company software defects prediction, *Information and Software Technology*, vol. 62, no. 1, pp. 67–77, 2015

[22] J. Zheng, Cost-sensitive boosting neural networks for software defect prediction, *Expert Systems with Applications*, vol. 37, no. 6, pp. 4537–4543, 2010.

[23] S. Wang and X. Yao, Using class imbalance learning for software defect prediction, *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[24] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, Cross-project defect prediction: A large scale experiment on data vs. domain vs. process, in *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Sigsoft Symposium on the Foundations of Software Engineering*, Amsterdam, the Netherlands, 2009, pp. 91–100.

[25] W. Dai, G. R. Xue, Q. Yang, and Y. Yu, Co-clustering-based classification for out-of-domain documents, in *Proc. of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Jose, CA USA, pp. 210–219, 2007.

[26] M. Muhammad,Y. Liu, M. Sun, and H. Luan, Enriching the transfer learning with pre-trained lexicon embedding for low-resource neural machine translation, *Tsinghua Science & Technology*, doi: 10.26599/TST.2020.9010029.

[27] J. Lin, L. Liang, X. Han, C. Yang, X. Chen, and X. Gao, Cross-target transfer algorithm based on the volterra model of SSVEP-BCI, *Tsinghua Science & Technology*, doi: 10.26599/TST.2020.9010015.

[28] Q. Wu, H. Wu, X. Zhou, M. Tan, Y. Xu, Y. Yan, and T. Hao, Online transfer learning with multiple homogeneous or heterogeneous sources, *IEEE Transactions on Knowledgeand Data Engineering*, vol. 29, no. 7, pp. 1494–1507, 2017.

[29] B. Turhan, T. Menzies, A. Bener, and J. Distefano, On the relative value of cross-company and within-company data for defect prediction, *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[30] I. H. Witten and E. Frank, Data mining: Practical machine learning tools and techniques, *Acm Sigmod Record*, vol. 31, no. 1, pp. 76–77, 2005.

[31] Y. Li, Z. Q. Huang, Y. Wang, and B. W. Fang, New approach of cross-project defect prediction based on multi-source data, (in Chinese), *Journal of Jilin University*, vol. 46, no. 6, pp. 2034–2041, 2015.

[32] C. Ma, LDA roaming guide, http://yuedu.baidu.com/ebook/d0b441a8ccbff121dd36839a.html, 2012

[33] G. Boetticher, T. Menzies, and T. Ostrand, The promise repository of empirical software engineering data, https://github.com/opensciences/opensciences.github.io, 2007.

[34] H. J. Ji and S. Huang, A new framework consisted of data preprocessing and classifier modelling for software defect prediction, *Computational Intelligence and Neuroscience*, vol. 2018, no. 1, pp. 1–13, 2018.

[35] H. Tong, B. Liu, and S. H. Wang, Transfer-learning oriented class imbalance learning for cross-project defect prediction, https://arxiv.org/abs/1901.08429, 2019.

[36] M. Shepperd, D. Bowes, and T. Hall, Researcher bias: The use of machine learning in software defect prediction, *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 603–616, 2014.

[37] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, Cliff's delta calculator: A non-parametric effect size program for two groups of observations, *Universitas Psychologica*, vol. 10, no. 2, pp 545–555, 2011.

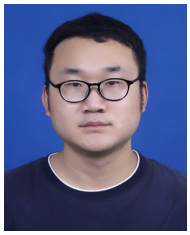
[38] Y. N. Wu, S. Huang, H. J. Ji, C. Y. Zheng, and C. Z. Bai, A novel Bayes defect predictor based on information diffusion function, *Knowledge-Based Systems*, vol. 144, no. 1, pp. 1–8, 2018.

[39] H. Tong, B. Liu, and S. H. Wang, Kernel spectral embedding transfer ensemble for heterogeneous defect prediction, *IEEE Transactions on Software Engineering*, doi: 10.1109/TSE.2019.2939303.

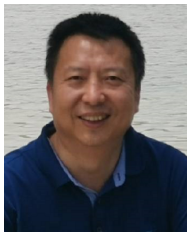
[40] Y. Freund and R. E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 23–37, 1997.

[41] S. Wang, H. Chen, and X. Yao, Negative correlation learning for classification ensembles, in *Proc. of International Joint Conference on Neural Networks*, Barcelona, Spain, 2010, pp. 1–8.

[42] I. H. Witten and E. Frank, Data mining: Practical machine learning tools and techniques, *Acm Sigmod Record*, vol. 31, no. 1, pp. 76–77, 2005.



Shiqi Tang received the MS degree in software engineering from Guilin University of Electronic Technology in 2017. He is currently a PhD candidate in software engineering at Army Engineering University of PLA. His research interests include fault localization, defect detection, and machine learning.



Song Huang received the PhD degree from PLA University of Science and Technology in 2002. He is a member of CCF and ACM. He is currently a professor at Software Testing and Evaluation Center, Army Engineering University of PLA. He is a member of the advisory boards of *Journal of Systems and Software*, *IEEE Transactions on Reliability*, etc. He has contributed more than 100 journal articles to professional journals. His current research interests include software testing, quality assurance, data mining, and empirical software engineering.

Changyou Zheng received the PhD degree in millitary information from PLA University of Science and Technology in 2013. He is currently a teacher at Army Enginerring University of PLA. His research interests include software testing and blockchains.

Erhu Liu received the BEng and MEng degrees from Southeast University in 2008 and 2013, respectively. He is a PhD candidate in software engineering at Army Engineering University of PLA. His research interests include AI testing and metamorphic testing.

Cheng Zong received the BEng degree in electric engineering and the MEng degree in control engineering from Southeast University, Nanjing, China in 2008 and 2014, respectively. He is currently a PhD candidate in software engineering at the Army Engineering University of PLA. His research interests include data mining, industrial control, and failure analysis.

Yixian Ding received the BEng degree in mechanical manufacturing engineering from Liaoning Technical University in 1990. He is currently working at Liaoning Technical University. His main research interest is application of project heat treatment in new materials.