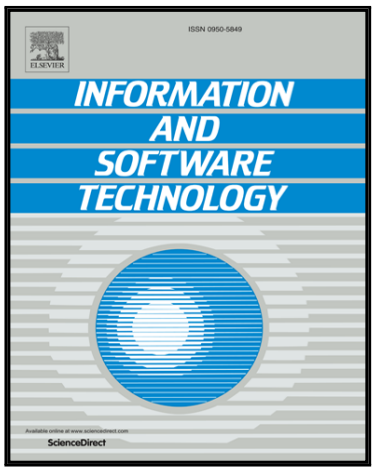


Software Defect Number Prediction: Unsupervised vs Supervised Methods

Xiang Chen, Dun Zhang, Yingquan Zhao, Zhanqi Cui, Chao Ni

PII: S0950-5849(18)30207-6
DOI: https://doi.org/10.1016/j.infsof.2018.10.003
Reference: INFSOF 6059

To appear in: *Information and Software Technology*

Received date: 6 February 2018
Revised date: 3 October 2018
Accepted date: 4 October 2018

Please cite this article as: Xiang Chen, Dun Zhang, Yingquan Zhao, Zhanqi Cui, Chao Ni, Software Defect Number Prediction: Unsupervised vs Supervised Methods, *Information and Software Technology* (2018), doi: https://doi.org/10.1016/j.infsof.2018.10.003

Software Defect Number Prediction: Unsupervised vs Supervised Methods

Xiang Chen[a,b,c,*], Dun Zhang[a], Yingquan Zhao[a], Zhanqi Cui[c,d], Chao Ni[c]

[a] *School of Computer Science and Technology, Nantong University, Nantong, China*
[b] *Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin, China*
[c] *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China*
[d] *Computer School, Beijing Information Science and Technology University, Beijing, China*

Abstract

Context: Software defect number prediction (SDNP) can rank the program modules according to the prediction results and is helpful for the optimization of testing resource allocation.

Objective: In previous studies, supervised methods vs unsupervised methods is an active issue for just-in-time defect prediction and file-level defect prediction based on effort-aware performance measures. However, this issue has not been investigated for SDNP. To the best of our knowledge, we are the first to make a thorough comparison for these two different types of methods.

Method: In our empirical studies, we consider 7 real open-source projects with 24 versions in total, use FPA and $Kendall$ as our effort-aware performance measures, and consider three different performance evaluation scenarios (i.e., within-version scenario, cross-version scenario, and cross-project scenario).

Result: We first identify two unsupervised methods with best performance. These two methods simply rank modules according to the value of metric LOC and metric RFC from large to small respectively. Then we compare 9 state-of-the-art supervised methods incorporating SMOTEND, which is used for handling class imbalance problem, with the unsupervised method based on LOC

*Corresponding author
Email addresses: xchencs@ntu.edu.cn (Xiang Chen), dunnzhang0@gmail.com (Dun Zhang), enockchao@gmail.com (Yingquan Zhao), czq@bistu.edu.cn (Zhanqi Cui), jacknichao920209@gmail.com (Chao Ni)

metric (i.e., LOC_D method). Final results show that LOC_D method can perform significantly better than or the same as these supervised methods. Later motivated by a recent study conducted by Agrawla and Menzies, we apply differential evolutionary (DE) to optimize parameter value of SMOTEND used by these supervised methods and find that using DE can effectively improve the performance of these supervised methods for SDNP too. Finally, we continue to compare LOC_D with these optimized supervised methods using DE, and LOC_D method still has advantages in the performance, especially in the cross-version and cross-project scenarios.

Conclusion: Based on these results, we suggest that researchers need to use the unsupervised method LOC_D as the baseline method, which is used for comparing their proposed novel methods for SDNP problem in the future.

Keywords: Software Defect Prediction, Software Defect Number Prediction, Supervised Method, Unsupervised Method, Class Imbalance Learning, Differential Evolutionary, Empirical Study

1. Introduction

Software defects are introduced unconsciously during the development process of software project. After the software project is deployed, defects in the software will produce unexpected behaviors, even will cause huge economic loss for enterprises in worst cases. Therefore project managers want to use software quality assurance methods (such as software testing, code inspection) to detect defects as many as possible. Due to the limitation of testing resources, project managers hope that they can resort to effective methods to identify potential defective modules as early as possible. Software defect prediction [1][2][3][4] is one of such effective methods. It constructs defect prediction models by mining software repositories (such as version control systems, bug tracking systems) and uses the constructed models to identify potential defective modules in new projects.

Most of previous studies discretize continuous defect number of program

modules into defective or non-defective [5]. This simple data preprocessing may lead to information loss. In addition, predicting defect number for program modules can assist in sorting program modules and then allocate more testing resources to these modules, which may contain more defects. In this way, the allocation of testing resources can be further optimized.

In previous studies, supervised methods vs unsupervised methods is an active issue for just-in-time defect prediction [6][7][8] and file-level defect prediction [9] based on effort-aware performance measures (such as ACC, P_{OPT}). To the best of our knowledge, we first make a thorough comparison for two different types of methods for software defect number prediction (SDNP) problem.

In our empirical studies, we consider 7 real open-source projects (such as ant, camel, etc) with 24 versions in total, use FPA and $Kendall$ as our performance measures, and consider three different performance evaluation scenarios (i.e., within-version scenario, cross-version scenario, and cross-project scenario). Final results show that some unsupervised methods can not be neglected in the future study for software defect number prediction.

To support the above conclusion, we mainly investigate three research questions in our empirical studies.

RQ1: Can supervised methods using SMOTEND perform better than unsupervised methods?

For this RQ, we first identify two unsupervised methods with best performance. These two methods rank modules according to the value of metric LOC and metric RFC from large to small, respectively. Then we compare 9 state-of-the-art supervised methods incorporating SMOTEND [10], which is used for solving class imbalance problem, with the unsupervised method based on LOC (i.e., LOC_D). Final results show that LOC_D can perform significantly better than or the same as these supervised methods in three scenarios.

RQ2: Can using differential evolutionary for SMOTEND improve the performance of supervised methods?

This RQ is motivated by a recent study conducted by Agrawla and Menzies [11]. We apply differential evolutionary (DE) for SMOTEND used by these

supervised methods and find that using DE can effectively improve the performance of these supervised methods. These findings are in consistent with the conclusions found by Agrawla and Menzies [11].

RQ3: Can supervised methods using DE perform better than unsupervised methods?

Based on the conclusion of RQ2, we continue to compare LOC_D with these supervised methods using DE for optimization, and find that previous conclusions in RQ1 still hold, especially in the cross-version and cross-project scenarios.

The main contributions of this paper can be summarized as follows:

- To the best of our knowledge, we are the first to consider unsupervised methods for SDNP problem and find that two unsupervised methods have best performance based on performance measures FPA and $Kendall$. These two methods simply rank modules according to the value of metric LOC and metric RFC from large to small, respectively.
- To thoroughly compare the performance of supervised methods and unsupervised methods, we consider three different validation scenarios (i.e., within-version scenario, cross-version scenario, and cross-project scenario).
- We apply differential evolutionary for SMOTEND to improve the performance of supervised methods for SDNP problem.
- Based on 7 real open-source projects, we conduct empirical studies to compare the performance of state-of-the-art supervised methods and unsupervised methods. Empirical results show that the unsupervised method LOC_D, which ranks modules according to the metric LOC from large to small, can perform significantly better than or the same as supervised methods, which are even optimized for SMOTEND by using differential evolutionary.

The rest of this paper is structured as follows: Section 2 summarizes related work. Section 3 shows all the supervised methods and unsupervised methods

used in our empirical studies. Section 4 shows experimental design, including experimental subjects, performance measures, performance evaluation scenarios, and statistical analysis methods used for result analysis. Section 5 and Section 6 perform result analysis and conduct some discussions. Section 7 analyzes threats to validity for our empirical studies. Section 8 concludes this paper and discusses some potential future work.

2. Related Work

In this section, we first review previous studies for software defect number prediction. Then we summarize the class imbalanced problem in software defect prediction. Finally, we review unsupervised methods for software defect prediction.

2.1. Software Defect Number Prediction

Predicting defect number for program modules can guide the sorting process of these modules and then allocate more testing resources to those program modules, which may contain more defects. In this way, more defects can be detected and repaired when given the limited testing resources.

Previous studies investigated regression based methods for this problem. Graves et al. [12] considered a generalized linear regression method. They conducted empirical studies on a large-scale telecommunication system and found that module's age, changes made to the modules, and the age of changes have a significant correlation. Wang and Zhang [13] proposed BugState, which was based on a defect state transition model. Ostrand et al. [14] employed negative binomial regression (NBR) method. Janes et al. [15] considered three methods (i.e., NBR, zero-inflated NBR, Poisson regression) for five real-time telecommunication systems. They found that zero-inflated NBR method achieved the best performance. Then Gao and Khoshgoftaar [16] further performed empirical studies on two industrial software projects and found that zero-inflated NBR can also achieve better performance.

Chen et al. [17] conducted empirical studies for six regression algorithms and found that using decision tree regression can achieve highest performance in both within-project scenario and cross-project scenario. Yu et al. [10] explored resampling (i.e., SMOTEND and RUSND) and ensemble learning (i.e., AdaBoost.R2) methods. Then they proposed two hybrid methods (i.e., SMOTENDBoost and RUSNDBoost) and these two methods can achieve higher performance.

Rathore and Kumar [18] explored the capability of decision tree regression in two different scenarios (i.e., intra-release prediction scenario and inter-release prediction scenario). They [19] compared six methods for SDNP, such as genetic programming, multi-layer perceptron, linear regression, decision tree regression, zero-inflated Poisson regression, and negative binomial regression. Recently, they [20][21] further considered ensemble learning methods for SDNP.

Based on the above analysis, we found that most of previous studies focus on regression based supervised methods and conduct empirical studies on these methods. However, none of previous studies investigate the possibility of unsupervised methods for SDNP.

2.2. Class Imbalance Problem for Software Defect Prediction

Datasets with imbalanced class distributions are quite common in many real applications, such as fraud detection, anomaly detection, medical diagnosis [22]. For software defect prediction, gathered datasets often have class imbalance problem too (i.e., the number of defective modules is overwhelmed by the number of non-defective modules). In this paper, the non-defective module is called as majority class while the defective module is called as minority class, since most of the defects are distributed in a small number of program modules. Most machine learning methods can have satisfactory performances when the number of instances for each class is roughly equal. When the number of instances for one class far exceeds the other, the performances of machine learning methods will be unsatisfactory. Many class imbalance methods have been proposed to alleviate this problem and these methods can be roughly classified into

three categories: sample based methods, cost-sensitive methods, and ensemble learning methods [22]. Until now, class imbalance problem for software defect prediction has been thoroughly investigated [23][24][25][26][27]. In a recent literature review [11], Chawla and Menzies found that most previous studies used SMOTE (synthetic minority over-sampling technique) [28], which is a typical sample based method, to solve the class imbalance problem for software defect prediction. Then they [11] proposed SMOTUNED, which is a self-tuning version of SMOTE by using differential evolutionary. Based on their empirical results, they concluded that data preprocessing was more important than classifier choice and SMOTUNED was a promising candidate for data preprocessing. Moreover SMOTUNED can even perform better than a new imbalance learning method MAHAKIL proposed by Bennin et al. [29]. In this paper, we will analyze whether their data preprocessing method [28] is applicable to SDNP problem.

2.3. Unsupervised Methods for Software Defect Prediction

It is not hard to find that most of previous studies focus on supervised methods. However, the expensive cost of gathering high-quality training set is still a barrier for applying software defect prediction to the real software development process. Therefore researchers started to investigate the possibility of unsupervised methods. The advantage of unsupervised methods can be summarized as follows: (1) It is straightforward to understand and much easier to implement. (2) It does not require any labeled training data or any complicated machine learning methods. Thus, unsupervised methods can be easily applied to a new project and have faster running speed.

Recently supervised methods vs unsupervised methods is a controversial issue for just-in-time defect prediction and file-level defect prediction based on effort-aware performance measures (such as ACC and P_{OPT}). In particular, for just-in-time (i.e., code change based) defect prediction, Kamei et al. [30] designed metrics based on diffusion, size, purpose, history, and developer experience for code changes. Then they proposed EALR method. Yang et al. [6]

later considered simple unsupervised methods and surprisedly found that some of these methods performed better than previous supervised methods in most cases when considering cross-validation, timewise-cross-validation, and cross-project-validation scenarios. Fu and Menzies [7] revisited Yang et al.'s empirical studies [6] and found that not all unsupervised methods had better performance than supervised methods. Therefore they proposed OneWay method, which can automatically select the potential best method. Huang et al. [8] also revisited Yang et al.'s study [6]. They considered two new performance measures (i.e., $PCI@20\%$ and IFA) and proposed a simple but improved method CBS. Liu et al. [31] proposed churn (i.e. the change size of a code change) based unsupervised method and found that this method performed better than previous proposed unsupervised methods [6]. Yan et al. [9] replicated Yang et al.'s study [6] for file-level defect prediction, which the granularity of the modules is set as file. They found that the conclusions of Yang et al. [6] did not hold under within-project defect prediction but held under cross-project defect prediction.

Different from previous studies, we want to investigate performance difference between supervised methods and unsupervised methods for SDNP problem.

3. Methods for Software Defect Number Prediction

In this section, we mainly introduce the supervised methods and unsupervised methods considered in our empirical studies for software defect number prediction.

3.1. Supervised Methods

In this subsection, we first introduce 9 state-of-the-art regression based supervised methods. Then we introduce the SMOTEND method used for solving class imbalance problem. Finally, we introduce how to use differential evolutionary to optimize the value of parameters used for SMOTEND.

3.1.1. Regression based Supervised Methods

Software defect number prediction is a typical regression problem. In this paper, we consider 9 state-of-the-art methods used in previous studies [10][17][5][19].

Linear regression (LR) is based on a statistical model. It is used to solve the least squares function of the linear relationship between one or multiple independent variables and a dependent variable.

Bayesian ridge regression (BRR) is based on a probabilistic model, which is similar to the Ridge Regression. The hyper parameters of such models are introduced by prior probability and then estimated by maximizing the marginal log likelihood with these probabilistic models.

Decision tree regression (DTR) is based on a decision tree model. In particular, it learns simple decision rules to approximate the curve of a given training set, and then predicts the target variable.

Nearest Neighbors Regression (NNR) is based on the k-nearest neighbor algorithm. The regression value of an instance is computed by the weighted average value of its nearest neighbors. Then the weight is set proportional to the inverse of the distance between the instance and its neighbors.

Gradient Boosting Regression (GBR) is in the form of an ensemble of weak prediction models. Several base learners are combined with a given machine learning method to improve the prediction performance over a single learner.

Random forest regressor (RF) is a meta estimator that fits a number of decision trees on various sub-samples of the dataset and uses averaging to improve the prediction performance and controls the over-fitting issue. The sub-sample size is always the same as the original dataset size, but the samples are drawn with replacement if bootstrap is set as true.

Moreover, we consider an ensemble learning method AdaBoost.R2 [32], which is a classical boosting algorithm for the regression problem. Based on the suggestions of the previous study [10], we consider DTR, BRR and LR as the base learner respectively and denote these supervised methods as ABR2_DTR, ABR2_BRR and ABR2_LR respectively.

3.1.2. SMOTEND method

However, the highly class imbalanced distribution may degrade the performance of SDNP methods [10]. In this paper, we consider SMOTEND [10] to handle this problem and make some modifications on this method.

Torgo et al. [33] mentioned three important issues to adapt SMOTE [28] for regression problem: (1) how to define the normal target variable values and the rare target variable values. (2) how to create new synthetic instances. (3) how to decide the target variable values of these new synthetic instances. For the first issue, we can define defective modules as rare instances and define non-defective modules as normal instances. For the second issue, we can use the same strategy of original SMOTE method. For the third issue, we can use a weighted average of the target variable values of the two seed instances. The weights are decided based on the distance between the synthetic instance and these two seed instances. The larger the distance is, the smaller the weight is.

In our SMOTEND, we identify three parameters: m, k and r. In particular, r is the parameter of Minkowski distance, while previous version of SMOTEND only considered Euclidean distance. Minkowski distance can be considered as a generalization of both the Euclidean distance and the Manhattan distance. The Minkowski distance with parameter r between two points $X = (x_1, x_2, \cdots, x_n)$ and $Y = (y_1, y_2, \cdots, y_n)$ can be computed as follows:

$$MD(X, Y, r) = (\sum_{i=1}^{n} |x_i - y_i|^r)^{1/r} \tag{1}$$

The parameter k determines the number of neighbors. The parameter m controls the number of synthetic instances. Different from previous version of SMOTEND [10], we use the function $NumSynthetic$ to compute the number of synthetic instances generated by SMOTEND. $NumSynthetic$ can be computed as follows:

$$NumSynthetic(m) = \frac{(|D_{maj}| - |D_{min}|) \times m}{N} \tag{2}$$

Here D_{maj} and D_{min} denote the majority instances (i.e., non-defective mod-

ules) and minority instances (i.e., defective modules), respectively. In the function $NumSynthetic$, we first compute the difference between $|D_{maj}|$ and $|D_{min}|$ and then divide it into N equal parts. For the convenience of analysis, we set N to 6 and use $m \in \{0, 1, \cdots, N\}$ as the candidate values for this parameter. When m is set as N, the training set D after using SMOTEND will have balanced class distributions. While when m is set as 0, it means that no synthetic instances are generated.

The pseudocode of SMOTEND can be found in Algorithm 1. It first computes the number of synthetic instances that need to be generated by using Function 2 (Line 1). Then it computes the nearest k neighbors for each instance in the minority class by using the Minkowski distance (Lines 4-6). Here $D_{min}[i]$ denotes the i-th instance for the minority class and $ins[i]$ stores the indexes of k neighbors for $D_{min}[i]$. Finally, it generates $numSynthetic$ synthetic instances (Lines 7-17). To generate a synthetic instance, it computes the difference of the feature vector between the i-th minority instance and its nearest neighbor, then it multiples this difference by a random number between 0 and 1, later it is added to the feature vector of the i-th minority instance. The number of defects for this synthetic instance is a weighted average of these two chosen instances. Here the $y(x)$ function returns the number of defects in the instance x.

3.1.3. SMOTEND Method Optimized by using Differential Evolutionary

The area of hyper parameter optimization for defect prediction models has been studies nowadays and researchers have found that the hyper parameter optimization can not be neglected in most cases [34][35][36][37][38]. In a recent study [11], Agrawal and Menizes proposed SMOTUNED, which is a self-tuning version of SMOTE by using differential evolutionary (DE) [39]. In their empirical studies, they found that SMOTUNED can effectively improve the performance and even perform better than a recently proposed method MAHAKIL [29]. However, SMOTUNED is optimized for traditional SMOTE method. In this paper, we will consider this optimization for our considered SMOTEND

Algorithm 1 SMOTEND

Input:

k - Number of Neighbours, m - Parameter Determining Number of Synthetic Instances, r - Parameter of Minkowski Distance, D_{min} - Instances of Minority Class D_{maj} - Instances of Majority Class

Output:

$D_{synthetic}$ - Synthetic Instances Generated by SMOTEND

$numSynthetic = \frac{(|D_{maj}|-|D_{min}|)\times m}{N}$ //N=6

$index = 0$

$D_{synthetic} = \emptyset$

for $i = 0; i < |D_{min}|; i++$ **do**

  Computing k nearest neighbours for the instance $D_{min}[i]$, and storing the indices in $ins[i]$ //using the Minkowski Distance

end for

while $numSynthetic > 0$ **do**

  $v_1 = D_{min}[index]$

  $v_2 = D_{min}[rand(ins[index])]$

  $new = v1 + rand(0, 1) \times (v_2 - v_1)$

  $d_1 = MD(new, v_1, r)$

  $d_2 = MD(new, v_2, r)$

  $y(new) = \frac{d_1 \times y(v_2) + d_2 \times y(v_1)}{d_1 + d_2}$

  $D_{synthetic} = D_{synthetic} \cup \{new\}$

  $index = (index + 1) \mod |D_{min}|$

  $numSynthetic - -$

end while

return $D_{synthetic}$

method.

Before introducing DE for SMOTEND, we first define some notations. A solution is comprised of D parameters and can be represented a vector (i.e., chromosome). For the sake of convenience, we call it the solution vector and D is the dimension of the vector. We use x_i^t to denote the i-th solution vector in the t-th generation and $x_{i,j}^t$ to denote the j-th value for this vector. For this optimization algorithm, D is set as three since we optimize three parameters (i.e., m, k, r) for SMOTEND. The function $fitness()$ is used to evaluate the quality (measured by FPA or $Kendall$, which will be introduced in Section 4) of the solution vector and larger value means better performance. For example, if we consider FPA as the performance measure, we first use SMOTEND by the parameter values based on the solution vector to generate synthetic instances and use the preprocessed dataset to train the model by a specific regression algorithm. Then we can use the prediction result of this model on the test set based on FPA as the fitness value.

The main process of this algorithm can be found in Algorithm 2. First, we initialize the first generation population. Here R_{max}^i and R_{min}^i denote the maximum and minimum value for the i-th parameter respectively. According to R and a random function $rand()$, we can randomly initialize each solution vector in the first generation (Lines 3-14). Here $bestParas$ represents the optimal solution vector found at present. If the generated new solution vector has better fitness value, $bestParas$ will be updated by this new solution vector (Lines 11-13).

Then the population will continually evolve (i.e., after T generations) and return $bestParas$ as the best solution vector (Lines 15-27). During each generation, we will use $genNewInstance$ to generate a new solution vector v, compare it to the vector x_j^{i-1} and choose the vector with larger fitness value into the next generation. Meanwhile, if the new vector x_j^i has the larger fitness value than $bestParas$, $bestParas$ will be updated by x_j^i.

We use Algorithm 3 to illustrate the function $genNewInstance$. This function first performs mutation operator. In particular, it randomly obtains three

vectors (v_1, v_2 and v_3) and creates a mutated vector (Lines 1-2). Here the type of two parameters for SMOTEND is integer, therefore we perform round operation on the value of corresponding parameters (Lines 3-7). Then the function performs crossover operator. In particular, we change some values of mutated vector to the values of the parent vector. Here $target$ denotes the parent vector, u denotes the mutated vector and v denotes the new vector, The crossover process can be found in Lines 8-15 and the judgment condition in Line 10 can make sure that at least one value in the mutated vector can go to the new vector.

The parameters and their search range of SMOTEND can be found in Table 1, including parameter name, default value, candidate values and its description. Moreover, the parameter value for differential evolutionary can be found in Table 2 based on the suggestions by Storn and Price [39].

Table 1: Parameters for SMOTEND

Parameter	Default Value	Candidate Values	Description
k	5	$\{1, 2, \cdots, 20\}$	Number of neighbors
m	6	$\{0, 1, \cdots, 6\}$	Determining the number of synthetic instances
r	2	$0.1 \leq r \leq 5$	Parameter for the Minkowski distance

Table 2: Parameter Value used for Differential Evolutionary [39]

Parameter	Value	Description
F	0.7	Differential Weight
CR	0.3	Crossover Probability
P	30	Population Size
T	8	Number of Generations
D	3	Number of Parameters

3.2. Unsupervised Methods

Unsupervised methods have attracted more interests in current software defect prediction research. These methods do not need any training set, are very

Algorithm 2 Differential Evolutionary for SMOTEND

Input:

P - Population Size, D - Number of Parameters, T - Number of Generations, R - Range of Parameters, F - Differential Weight, CR - Crossover Probability

Output:

$bestParas$ - Best Solution Vector

$bestParas = NULL$
$t = 0$
{Population initialization}
for $i = 0; i < P; i++$ **do**
  for $j = 0; j < D; j++$ **do**
    if The type of $j-th$ parameter is continuous **then**
      $x_{i,j}^t = R_{min}^j + rand(0,1) \times (R_{max}^j - R_{min}^j)$
    else if The type of $j-th$ parameter is integer **then**
      $x_{i,j}^t = R_{min}^j + [rand(0,1) \times (R_{max}^j - R_{min}^j)]$
    end if
  end for
  if $fitness(bestParas) < fitness(x_i^t)$ **then**
    $bestParas = x_i^t$
  end if
end for
{Population evolution}
for $i = 1; i < T; i++$ **do**
  for $j = 0; j < P; j++$ **do**
    $v = genNewInstance(D, F, CR, x_j^{i-1}, i)$
    if $fitness(v) > fitness(x_j^{i-1})$ **then**
      $x_j^i = v$
    else
      $x_j^i = x_j^{i-1}$
    end if
    if $fitness(bestParas) < fitness(x_j^i)$ **then**
      $bestParas = x_j^i$
    end if
  end for
end for
return $bestParas$

Algorithm 3 genNewInstance

Input:

D - Number of Parameters, F - Differential Weight, CR - Crossover Probability, $target$ - The Vector Solution used for Crossover the Mutation, i - i-th Generation

Output:

v - The New Generated Solution Vector

{Performing mutation operator}

Selecting three vectors x_{t1}^{i-1},x_{t2}^{i-1},x_{t3}^{i-1} randomly and $(t1 \neq t2 \neq t3)$

u=$x_{t1}^{i-1} + F \times (x_{t2}^{i-1} - x_{t3}^{i-1})$

for $k = 0$; $k < D$; $k++$ **do**

  if The type of k-th parameter is integer **then**

    $u_k = [u_k]$

  end if

end for

{Performing crossover operator}

$rd = randint(D)$

for $k = 0$; $k < D$; $k++$ **do**

  if $rand(0, 1) < CR$ or $k == rd$ **then**

    $v_k = u_k$

  else

    $v_k = target_k$

  end if

end for

return v

simple, and have a low model construction cost. Yang et al. [6] proposed 12 simple unsupervised methods based on metrics (such as NS, ND, etc) for just-in-time (i.e, code change based) defect prediction. For a specific metric, it computes the defect-proneness probability for i-th code change as $1/v_i$, where v_i denotes the value of i-th code change for this metric. This means that the smaller the metric value, the higher the defect-proneness probability. Then all the code changes will be ranked in the descendant order according to the computed probability. The idea of these unsupervised methods [6] is motivated by the findings of Koru et al. [40][41] that smaller modules should be inspected first, since the relationship between the module size and the number of defects is logarithmic. ManualUp model (i.e., smaller modules should be inspected first) proposed by Menzies et al. [42] further verified Koru et al.'s findings. However, the effectiveness of these unsupervised methods is only verified on two effort-aware performance measures (i.e., ACC and P_{OPT}). For software defect number prediction, we mainly consider two different performance measures (i.e., FPA and $Kendall$), which will be introduced in Section 4. Therefore, we still consider the simple unsupervised methods proposed by Yang et al. [6] and Yan et al. [9]. That is to say, given a metric, we rank the program modules in the ascending order according to the metric value. Moreover, we further consider unsupervised methods using another ranking strategy. That is to say, we rank the program modules in the descending order according to the metric value when given a metric. Since our subjects consider 20 code complexity based metrics, which will be introduced in Section 4, 40 different simple unsupervised methods in total are used in our empirical studies.

4. Experimental Design

4.1. Experimental Subjects

In our empirical studies, we choose 7 experimental subjects (with 24 versions in total) from open-source projects. These subjects can be downloaded from

seacraft repository [1] and they are widely used in previous empirical studies [43][9][17][10][20][21][44].

The granularity of the program modules in these subjects is set as class. The characteristics of these subjects are shown in Table 3, which includes project name, project version, number of modules, number of defective modules, and the maximum defects contained in the modules.

For these subjects, metrics are designed based on the code complexity and features of object oriented program [43]. Table 4 includes metric category, metric name and corresponding description. Compared to software development process based metrics, the value of these metrics can be automatically extracted from program modules with a few efforts even for large-scale software systems [45].

4.2. Performance Measures

For software defect number prediction, AAE (average absolute error), ARE (average relative error), and $RMSE$ (root mean square error) are often used in previous studies [17][20][21]. However these measures can result in over-optimistic estimation when datasets have the class imbalance problem. Supposing we have a simple model, this model can predict all the modules as non-defective (i.e., the defect number for each module is 0). If we use this model to make a prediction on dataset Camel-1.0, which contains 14 defective modules and 325 non-defective modules, AAE value of this model is only 14/339. However this model is useless since they can not identify any defective modules. Therefore in this paper, we consider FPA [46] and $Kendall$ [47] measures used by Yu et al. [10].

Kendall rank correlation coefficient ($Kendall$ for short) is a statistic used to measure the ordinal association between two measured quantities. A higher Kendall coefficient means a better ranking. Let $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$ be a set of observations of the joint random variables X and Y respectively.

[1] http://tiny.cc/seacraft

Table 3: Statistic of Experimental Subjects

Project	Version	#Modules	#Defective Modules	%Defective Modules	Max Defects
ant	ant-1.3	125	33	16.00%	3
	ant-1.4	178	47	22.47%	3
	ant-1.5	293	35	10.92%	2
	ant-1.6	351	184	26.21%	10
	ant-1.7	745	338	22.28%	10
camel	camel-1.0	339	14	3.83%	2
	camel-1.2	608	522	35.53%	28
	camel-1.4	872	335	16.63%	17
	camel-1.6	965	500	19.48%	28
ivy	ivy-1.1	111	233	56.76%	36
	ivy-1.4	241	18	6.64%	3
	ivy-2.0	352	56	11.36%	3
jedit	jedit-3.2	272	382	33.09%	45
	jedit-4.0	306	226	24.51%	23
	jedit-4.1	312	217	25.32%	17
	jedit-4.2	367	106	13.08%	10
synapse	synapse-1.0	157	21	10.19%	4
	synapse-1.1	222	99	27.03%	7
	synapse-1.2	256	145	33.59%	9
xalan	xalan-2.4	723	156	15.21%	7
	xalan-2.5	803	531	48.19%	9
	xalan-2.6	885	625	46.44%	9
xerces	xerces-1.2	440	115	16.14%	4
	xerces-1.3	453	193	15.23%	30

Table 4: Metrics used by Experimental Subjects

Category	Metric Name	Description
Complexity	LOC	Lines of Code
	WMC	Weighted Methods per Class
	NPM	Number of Public Methods
	AMC	Average Method Complexity
	Max_cc	Max McCabe's Cyclomatic Complexity
	Avg_cc	Avg McCabes Cyclomatic Complexity
	MOA	Measure of Aggregation
Coupling	CBO	Coupling between Object Classes
	RFC	Response for a Class
	CA	Afferent Couplings
	CE	Efferent Couplings
	IC	Inheritance Coupling
	CBM	Coupling Between Methods
Cohesion	LCOM	Lack of Cohesion in Methods
	LCOM3	Lack of Cohesion in Methods
	CAM	Cohesion Among Methods of Class
Abstraction	DIT	Depth of Inheritance Tree
	NOC	Number Of Children
	MFA	Measure of Functional Abstraction
Encapsulation	DAM	Data Access Metric

Here x_i and y_i denote the actual number of defects and the predicted number of defects in i-th module respectively. Any pair of observations (x_i, y_i) and (x_j, y_j), where $i \neq j$, is defined to be concordant if the ranks for both elements agree (i.e., both $x_i > x_j$ and $y_i > y_j$ or both $x_i < x_j$ and $y_i < y_j$). The pair is said to be discordant, if both $x_i > x_j$ and $y_i < y_j$ or both $x_i < x_j$ and $y_i > y_j$. If $x_i = x_j$ or $y_i = y_j$, the pair is neither concordant nor discordant. The Kendall coefficient τ can be defined as:

$$\tau = \frac{\#\text{concordant pairs} - \#\text{discordant pairs}}{n(n-1)/2} \tag{3}$$

FPA (Fault-Percentile-Average) was previously proposed by Weyuker et al. [46]. Supposing that the ranking results of the k modules for a specific method are f_1, f_2, $\cdots$, f_k. If the i-th module has n_i defects, the total number of defects are $n = n_1 + n_2 + \cdots + n_k$. The proportion of the actual defects in the top m predicted modules to the whole defects is:

$$p(m) = \sum_{i=k-m+1}^{k} \frac{n_i}{n} \tag{4}$$

Then FPA can be computed as:

$$FPA = \frac{1}{k} \sum_{m=1}^{k} p(m) \tag{5}$$

It is not hard to find that a higher FPA means a better ranking, which the modules with more defects are ranked in the top.

4.3. Performance Evaluation Scenarios

In a recent study [10], Yu et al. merged the different versions of the same project as a dataset and then used 10-fold cross-validation to evaluate the performance of SDNP methods. However, this experimental setting is not reasonable, since randomly partitioning the dataset may cause a model to use future knowledge, which should not be known at the time of model construction, to predict modules in the past [23].

In this paper, we consider three different performance evaluation scenarios (i.e., within-version, cross-version, and cross-project). These three scenarios do not have the issue in the previous study [10] and these scenarios are used in previous SDNP studies [44][17][20][19][21]. In particular, (1) In the within-version defect prediction scenario, we considers 5 × 5-fold cross validation. In particular, for a target version of a given project, we split the dataset into 5 folds of approximately equal size and each fold has a similar class distribution. 4 folds are used as the training set to train the model, while the remaining fold is used as the test set to test the performance of the model. This cross-validation is repeated 5 times so that each fold is used exactly once as the test data. The entire 5-fold cross validation process is then repeated 5 times to alleviate possible sampling bias in random splits [2]. (2) For a target version of a given project, the cross-version defect prediction scenario uses data from the versions developed before the current version in the same project as the training set. (3) For a target version of a given project, the cross-project defect prediction uses data from all the versions of another project as the training set. Notice that SMOTEND method or SMOTEND method optimized by using differential evolutionary is only applied to the training set and not to the test set.

We use an example shown in Figure 1 to illustrate these three different scenarios. If we use ant-1.7 as the test set (i.e., the target version), for the within-version defect prediction, we consider 5 × 5-fold cross validation. For each split, we use 80% as the training set and the remaining 20% as the test set. This split process is repeated 25 times. For the cross-version scenario, we use data set based on ant-1.3 to ant-1.6 as the training set. For the cross-project scenario, if we choose xerces project, we will use datasets based on xerces-1.2 to xerces-1.3 as the training set. Obviously, we can choose all the versions of other project (such as camel, ivy, etc) as the training set in this scenario.

In our empirical studies, we do not consider all the versions as our target version. First we do not consider the first version for each project since they can

[2]Notice that the random number seed is different for each dataset split.

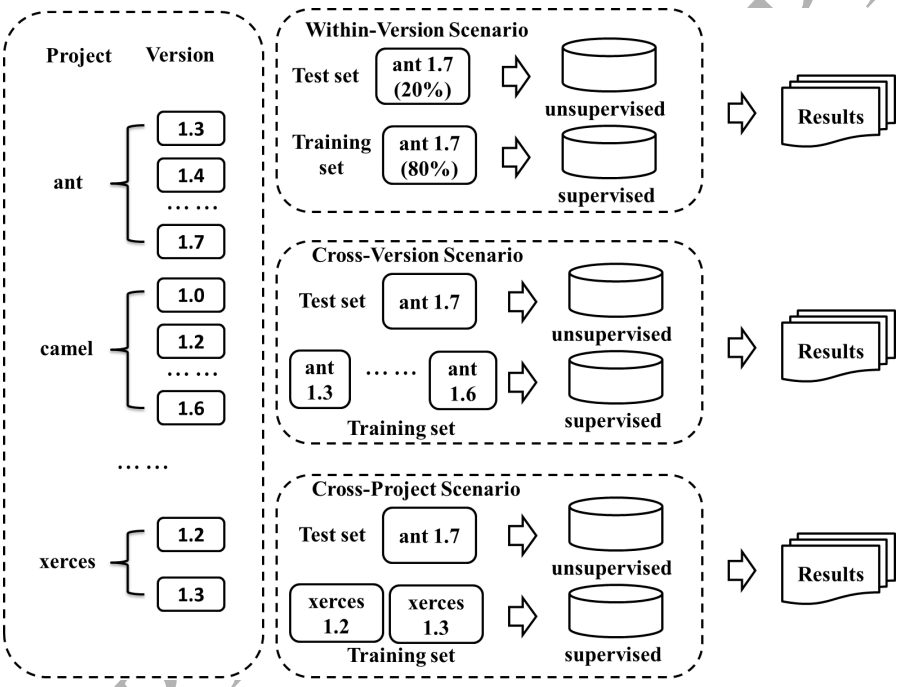


Figure 1: Example for Illustrating Three Different Performance Evaluation Scenarios

not be used for the cross-version scenario. Then we do not choose the versions, whose class imbalance rate is too low, such as ivy-1.4. Finally, we choose 15 versions as our target versions. In particular, For ant project, we choose three versions (i.e., 1.5, 1.6 and 1.7). For camel project, we choose three versions (i.e., 1.2, 1.4 and 1.6). For ivy project, we only choose 2.0 version. For jedit project, we choose three versions (i.e., 4.0, 4.1 and 4.2). For synapse project, we choose two versions (i.e., 1.1 and 1.2). For xalan project, we choose two versions (i.e., 2.5 and 2.6). For xerces, we only choose 1.3 version.

4.4. Statistical Analysis Method

To rank all the different supervised methods and unsupervised methods, we use Scott-Knott test, which is recommended by Ghotra et al. [48]. Scott-Knott test is used to examine whether some methods outperform others and generates a global ranking of these methods. In particular, Scott-Knott test performs the grouping process in a recursive way. Firstly, Scott-Knott test uses a hierarchical cluster analysis to partition all the methods into two ranks based on the mean performance (FPA or $Kendall$). Then if the divided ranks are significantly different, Scott-Knott test is recursively executed again within each rank to further divide the ranks. When ranks can no longer be divided into statistically distinct ranks, the test will terminate.

We use Wilcoxon signed-rank test to examine whether the performance difference between two methods are statistically significant. We also use the Benjamini-Hochberg (BH) procedure to adjust p-values if we perform multiple comparisons. Then if the test shows a significant difference, we compute Cliffs δ, which is a non-parametric effect size measure, to examine whether the magnitude of the difference is substantial or not. The meaning of different Cliffs δ value and their corresponding interpretation are shown in Table 5. In summary, a method performs significantly better or worse than another method, if BH corrected p-value is less than 0.05 and the effectiveness level is not negligible based on Cliffs δ. While the difference between two methods is not significant, if p-value is not less than 0.05 or p-value is less than 0.05 and the effectiveness

level is negligible.

Table 5: Cliff's δ and Corresponding Effectiveness Level [49]

Cliff's δ	Effectiveness Level
$\|\delta\| < 0.147$	Negligible
$0.147 \leq \|\delta\| < 0.33$	Small
$0.33 \leq \|\delta\| < 0.474$	Medium
$0.474 \leq \|\delta\|$	Large

5. Result Analysis

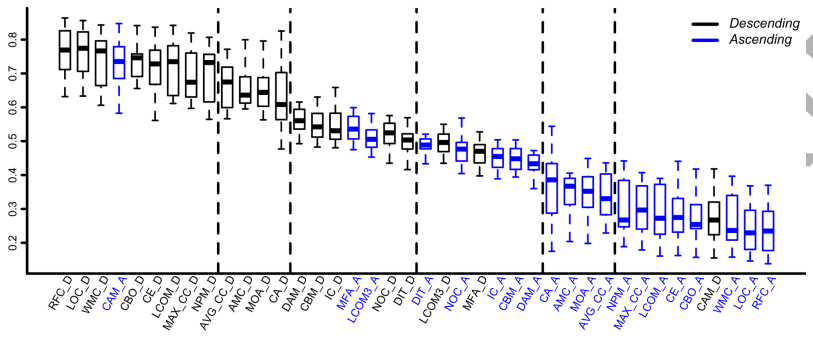
5.1. Result Analysis for RQ1

RQ1: Can supervised methods using SMOTEND perform better than unsupervised methods?

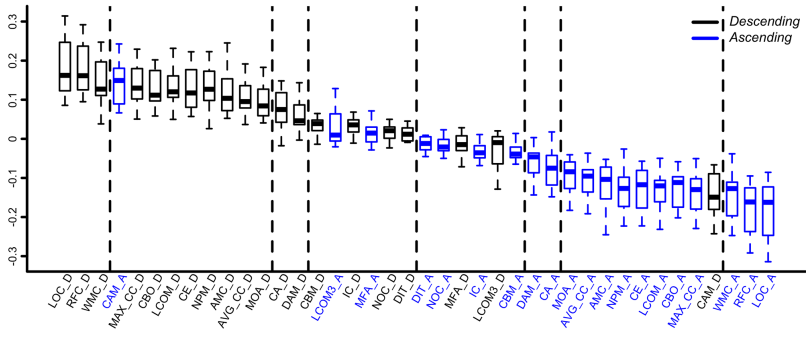
To answer this RQ, we first identify these unsupervised methods with best performance. As discussed in Section 3, we considered 40 unsupervised methods in total by considering two different ranking strategies. Here the method name taking A (Ascending) as the suffix indicates the modules are ranked according to the corresponding metric value from small to large. In this way, the modules with smaller metric value will be ranked higher. While the method name taking D (Descending) as the suffix indicates the modules are ranked according to the corresponding metric value from large to small. In this way, the modules with larger metric value will be ranked higher. Considering the cross-version scenario [3], we use Scott-Knott test to group these methods into statistically distinct ranks. The results are shown in Figure 2. The dotted lines represent groups divided by using the Scott-Knott test. All methods are ordered based on their mean ranks. The distribution of FPA and $Kendall$ is shown using

[3]The results are same for the cross-project scenario, since these two scenarios use all the data of the target version as the test data.

boxplot [4]. The blue label denotes unsupervised methods using the ascending strategy and the black label denotes unsupervised methods using the descending strategy.



(a) FPA



(b) $Kendall$

Figure 2: Comparison Results of Different Unsupervised Methods in the Cross-Version Scenario based on Scott-Knott Test

It is not hard to find that when considering FPA and $Kendall$ performance measures, unsupervised methods using the descending ranking strategy perform better than unsupervised methods using the ascending ranking strategy. These

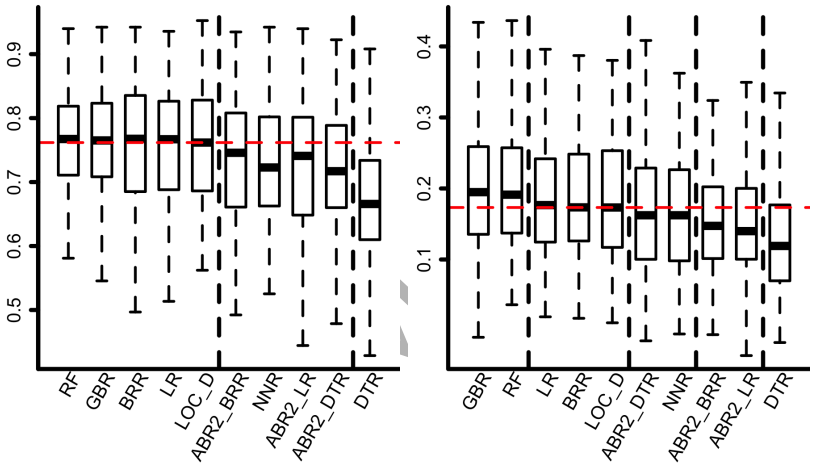
[4]A boxplot consists of five most important sample percentiles: the sample minimum, the lower quartile, the median, the upper quartile and the sample maximum.

findings are different from the findings in studies of just-in-time defect prediction and file-level defect prediction when considering effort-aware performance measures [6][9]. For these unsupervised methods using the descending ranking strategy, we further identify two methods (i.e., RFC_D and LOC_D), which can achieve the best performance. When using Wilcoxon signed-rank test to compare RFC_D method with LOC_D method, the BH corrected p-Value is 0.72 and 0.91 respectively for FPA and $Kendall$ measures. It shows that there is no significant difference in performance between these two methods. Similar results can be also found in the within-version scenario. Since the LOC metric for the program modules can be more easily measured than RFC metric, we choose LOC_D as the representative method for unsupervised learning methods and then compare this method with state-of-the-art supervised methods.

In three different scenarios, we all use SMOTEND to solve class imbalance problem in the datasets. Moreover, in the cross-project defect prediction scenario, we consider Burak filter method [50] to alleviate the distribution difference between the source project and the target project [51]. In particular, Burak filter method [50] can choose relevant modules according to the characteristic of the target project. The results of the Scott-Knott test between LOC_D and these 9 supervised methods can be found in Figure 3, Figure 4 and Figure 5 in three different scenarios. In these figures, the horizontal red dashed lines indicate the median value of the LOC_D method, which is to help visualize the median differences between LOC_D and different supervised methods.

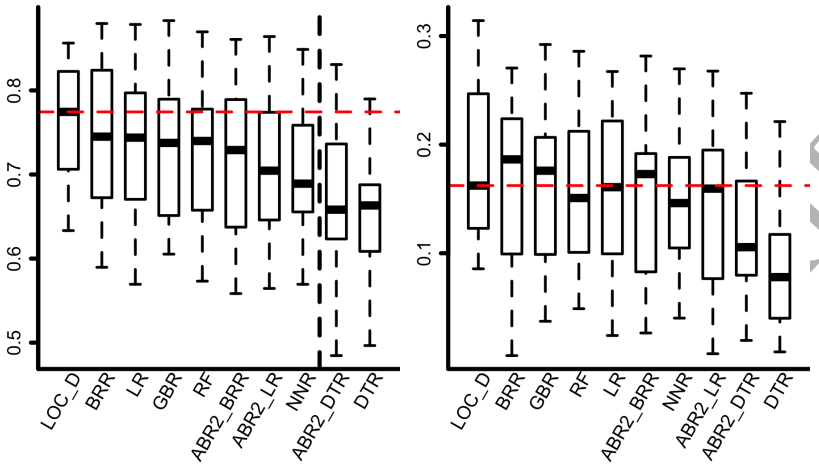
Based on these figures, we can find that in the within-version scenario when considering FPA, RF method performs best, however, LOC_D and RF are in the same group. While in the cross-version scenario and the cross-project scenario, LOC_D surprisedly performs best. The advantage is more obvious in the cross-project scenario.

The detailed comparison results between unsupervised method LOC_D and supervised methods can be found in Table 6, Table 7 and Table 8 for three different scenarios. These tables show the mean $Kendall$ and FPA, Win/Draw/Loss (W/D/L) results based on $Kendall$ and FPA, and # top ranks based on



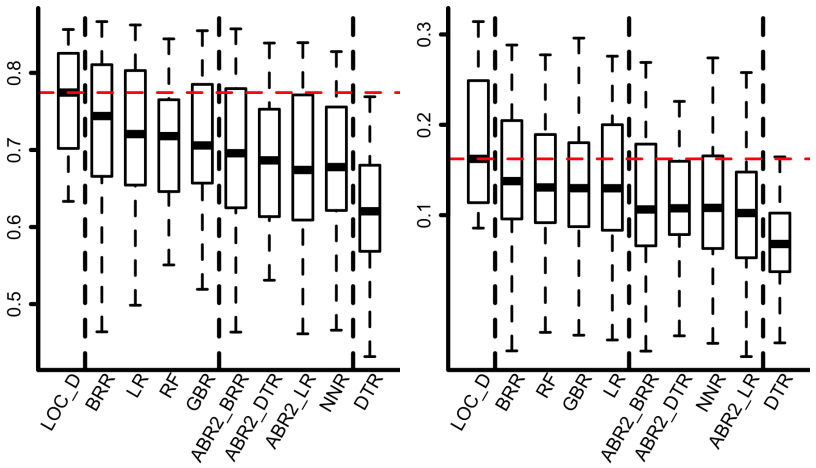
(a) FPA (b) $Kendall$

Figure 3: Comparison Result between Supervised Methods and Unsupervised Method LOC_D in the Within-version Defect Prediction Scenario based on Scott-Knott Test, Notice the Larger the $FPA/Kendall$ Value, the Better the Performance of the Method



(a) FPA (b) $Kendall$

Figure 4: Comparison Result between Supervised Methods and Unsupervised Method LOC_D in the Cross-version Defect Prediction Scenario based on Scott-Knott Test



(a) FPA (b) $Kendall$

Figure 5: Comparison Results between Supervised Methods and Unsupervised Method LOC_D in the Cross-project Defect Prediction Scenario based on Scott-Knott Test

Kendall and *FPA*. In the within-version scenario, W/D/L results show the number of datasets, on which the corresponding method performs significantly better than, the same as, worse than the LOC_D method by using Wilcoxon signed-rank test, since 5 × 5-fold cross validation is applied for each dataset. In the remaining two scenarios, W/D/L results show the number of cases, on which the corresponding method performs better than, the same as, worse than the LOC_D method. # top ranks can count #top ranks for each method. Notice in some cases, there may exist multiple methods in the top rank. Finally, we use Wilcoxon signed-rank test to examine whether the performance difference between the corresponding method and LOC_D method is statistically significant. The BH corrected *p*-Value is shown in rows *p*-value (*Kendall*) and *p*-value (*FPA*). If *p*-Value is smaller than 0.05, we further analyze Cliff's δ and set a shadow on the cell if its value is negligible.

(1) In the within-version scenario, when considering mean *Kendall* value, LOC_D performs better than 55.6% (5/9) supervised methods. The mean *Kendall* value of LOC_D is 0.184. The best method is GBR and its mean *Kendall* value is 0.203. Based on W/D/L (*Kendall*) analysis, LOC_D achieves at least 50% wins (or draws) when comparing to all the supervised methods. In most cases, LOC_D achieves at least 66.7% (10/15) wins (or draws). When counting #top ranks (*Kendall*), GBR is in the first place, which can achieve top rank in 5 cases. LOC_D and RF are in the second place, which can achieve top rank in 4 cases. When considering mean *FPA* value, LOC_D performs better than 55.6% (5/9) supervised methods. The mean *FPA* value of LOC_D is 0.757. The best method is RF and its mean *FPA* value is 0.765. Based on W/D/L (*FPA*) analysis, LOC_D achieves at least 50% wins (or draws) when comparing to all the supervised methods. In most cases, LOC_D achieves at least 66.7% (10/15) wins (or draws). When counting #top ranks (*FPA*), both LOC_D and GBR are in the first place, which can achieve top rank in 4 cases. Based on BH corrected *p*-Value and Cliffs δ, we find that based on *Kendall*, LOC_D performs significantly better than 3 supervised methods (i.e., DTR, ABR2_LR and ABR2_BRR). While the remaining supervised methods do not

perform significantly better than LOC_D. Based on FPA, LOC_D performs significantly better than 4 supervised methods (i.e., DTR, NNR, ABR2_LR and ABR2_DTR). While the remaining supervised methods do not perform significantly better than LOC_D.

(2) In the cross-version scenario, when considering mean $Kendall$ value, LOC_D performs better than all the supervised methods. The mean $Kendall$ value of LOC_D is 0.182. Based on W/D/L ($Kendall$) analysis, LOC_D achieves at least 50% wins (or draws) when comparing to all the supervised methods. In most cases, LOC_D achieves at least 60% (9/15) wins (or draws). When counting #top ranks ($Kendall$), LOC_D is in the first place, which can achieve top rank in 6 cases. GBR is in the second place, which can achieve top rank in 5 cases. When considering mean FPA value, LOC_D performs better than all the supervised methods. The mean FPA value of LOC_D is 0.759. Based on W/D/L (FPA) analysis, LOC_D achieves at least 50% wins (or draws) when comparing to all the supervised methods. In most cases, LOC_D achieves at least 60% (9/15) wins (or draws). When counting #top ranks (FPA), both LOC_D and GBR are in the first place, which can achieve top rank in 6 cases. Based on BH corrected p-Value and Cliffs δ, we find that based on $Kendall$, LOC_D performs significantly better than 2 supervised methods (i.e., DTR and ABR2_DTR). While the remaining supervised methods do not perform significantly better than LOC_D. Based on FPA, LOC_D performs significantly better than 3 supervised methods (i.e., DTR, NNR and ABR2_DTR). While the remaining supervised methods do not perform significantly better than LOC_D.

(3) In the cross-project scenario, when considering mean $Kendall$ value, LOC_D performs better than all the supervised methods. The mean $Kendall$ value of LOC_D is 0.182. Based on W/D/L ($Kendall$) analysis, LOC_D achieves at least 50% wins (or draws) when comparing to all the supervised methods. In most cases, LOC_D achieves at least 74.4% (67/90) wins (or draws). When counting #top ranks ($Kendall$), LOC_D is in the first place, which can achieve top rank in 64 cases. When considering mean FPA value, LOC_D performs better than all the supervised methods. The mean FPA value of LOC_D is

0.759. Based on W/D/L (FPA) analysis, LOC_D achieves at least 50% wins (or draws) when comparing to all the supervised methods. In most cases, LOC_D achieves at least 70% (63/90) wins (or draws). When counting #top ranks (FPA), both LOC_D and GBR are in the first place, which can achieve top rank in 57 cases. Based on BH corrected p-Value and Cliffs δ, we find LOC_D performs significantly better than all the supervised methods based on whether $Kendall$ or FPA.

> Unsupervised Method LOC_D can perform significantly better than or the same as state-of-the-art supervised methods using SMOTEND.

5.2. Result Analysis for RQ2

RQ2: Can using differential evolutionary for SMOTEND improve the performance of supervised methods?

For this RQ, we first analyze whether the tuned parameter values are different from the default parameter values. Then we want to analyze whether there is any benefit in tuning the parameters for SMOTEND by using differential evolutionary.

As discussed before, the default values for parameters (i.e., k, m, r) of SMOTEND are 5, 6, 2 respectively. Figure 6 shows the distribution of values for these parameters in the cross-version scenario using boxplot when optimized for FPA measure. It is not hard to find that most of tuned parameter values are different from their default values. For example, median of parameter k's value is larger than 5 in most cases. Therefore, using default value for parameters of SMOTEND is not recommended for SDNP problem either. Similar results can be also found in other two performance evaluation scenarios.

Figure 7 shows the performance improvement ratio when ivy-2.0 is set as the target version in the cross-project scenario when considering both $Kendall$ measure and FPA measure. For each subfigure, x-axis shows different supervised methods and y-axis shows performance improvement ratio computed by $(value_{optimize} - value_{orignial})/value_{orignial}$. Here $value_{optimize}$ denotes the per-

Table 6: Comparison Results between Unsupervised Method LOC_D with Supervised Methods in the Within-version Defect Prediction Scenario

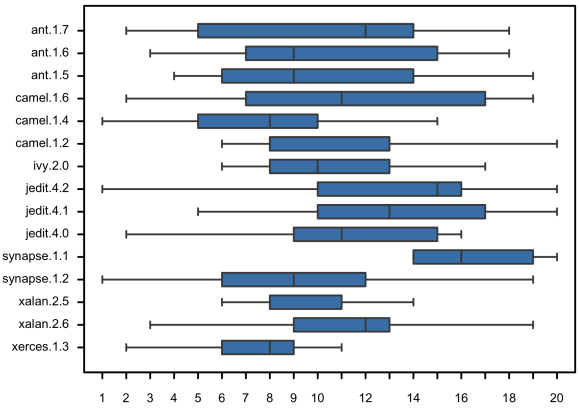
	LR	BRR	DTR	NNR	GBR	RF	ABR2_LR	ABR2_DTR	ABR2_BRR	LOC_D
Kendall	0.186	0.186	0.126	0.167	**0.203**	0.201	0.150	0.167	0.153	0.184
W/D/L (*Kendall*)	5/6/4	3/10/2	0/3/12	2/4/9	7/3/5	7/3/5	2/5/8	2/4/9	2/5/8	
#top ranks (*Kendall*)	1	1	0	0	5	4	0	0	1	4
p value (*Kendall*)	6.34E-01	6.48E-01	1.09E-19	7.63E-03	4.12E-03	8.25E-03	2.09E-07	1.02E-02	4.16E-06	
FPA	0.757	0.758	0.672	0.729	0.763	**0.765**	0.727	0.721	0.734	0.757
W/D/L (*FPA*)	3/7/5	3/10/2	0/0/15	1/2/12	5/7/3	7/5/3	1/6/8	0/8/7	2/4/9	
#top ranks (*FPA*)	3	2	0	0	4	2	1	0	1	4
p value (*FPA*)	8.52E-01	7.07E-01	7.86E-31	2.02E-05	3.61E-01	2.50E-01	8.99E-05	8.32E-07	2.56E-03	

Table 7: Comparison Results between Unsupervised Method LOC_D with Supervised Methods in the Cross-version Defect Prediction Scenario

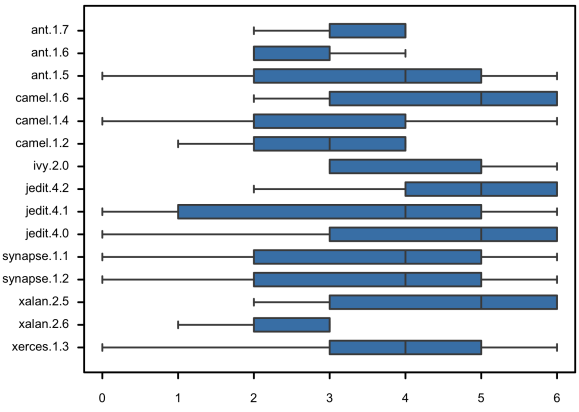
	LR	BRR	DTR	NNR	GBR	RF	ABR2_LR	ABR2_DTR	ABR2_BRR	LOC_D
Kendall	0.156	0.165	0.094	0.147	0.164	0.159	0.143	0.123	0.149	**0.182**
W/D/L (*Kendall*)	4/1/10	6/0/9	1/1/13	3/1/11	7/0/8	5/0/10	4/0/11	3/0/12	6/0/9	
#top ranks (*Kendall*)	0	0	0	0	5	2	1	0	1	6
p value (*Kendall*)	3.30E-01	5.20E-01	2.64E-03	2.72E-01	6.04E-01	4.43E-01	2.21E-01	4.01E-02	3.73E-01	
FPA	0.734	0.744	0.656	0.705	0.730	0.729	0.709	0.672	0.720	**0.759**
W/D/L (*FPA*)	6/0/9	6/0/9	0/0/15	1/0/14	7/0/8	5/0/10	2/0/13	1/0/14	6/0/9	
#top ranks (*FPA*)	1	1	0	1	6	1	0	0	0	6
p value (*FPA*)	4.68E-01	7.56E-01	1.13E-03	8.52E-02	3.10E-01	3.10E-01	1.01E-01	2.13E-02	2.72E-01	

Table 8: Comparison Results between Unsupervised Method LOC_D with Supervised Methods in the Cross-project Defect Prediction Scenario

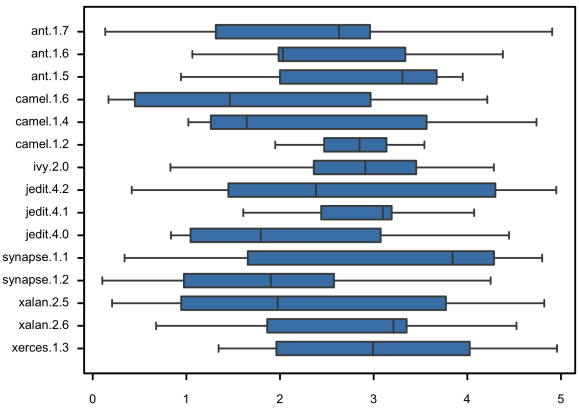
	LR	BRR	DTR	NNR	GBR	RF	ABR2_LR	ABR2_DTR	ABR2_BRR	LOC_D
Kendall	0.134	0.147	0.072	0.113	0.135	0.137	0.108	0.116	0.117	**0.182**
W/D/L (*Kendall*)	13/2/75	22/1/67	0/1/89	0/0/90	4/0/86	2/1/87	5/0/85	1/0/89	6/1/83	
#top ranks (*Kendall*)	8	17	0	0	1	2	0	0	1	64
p value (*Kendall*)	3.27E-05	2.82E-03	6.02E-21	6.55E-09	4.74E-05	6.35E-05	7.16E-10	3.91E-09	2.19E-08	
FPA	0.718	0.730	0.622	0.682	0.710	0.711	0.683	0.685	0.692	**0.759**
W/D/L (*FPA*)	16/0/74	26/3/61	0/0/90	2/0/88	2/0/88	6/0/84	5/1/84	0/1/89	9/2/79	
#top ranks (*FPA*)	8	21	0	2	1	3	1	0	1	57
p value (*FPA*)	2.25E-03	3.78E-02	5.98E-22	2.58E-09	1.92E-05	3.79E-05	6.40E-08	2.58E-09	8.37E-07	



(a) Tuned Value Distribution for Parameter k



(b) Tuned Value Distribution for Parameter m



(c) Tuned Value Distribution for Parameter r

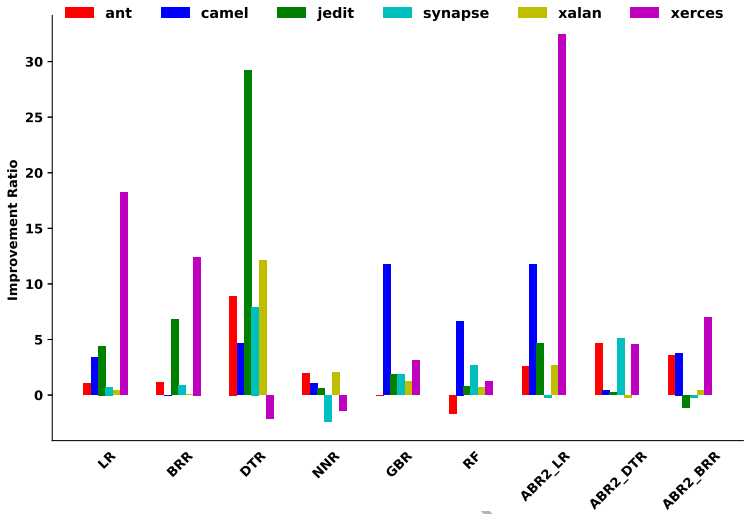
Figure 6: Tuned Value Distribution for Three Parameters of SMOTE after using DE When Optimized in the Cross-version Scenario for FPA Measure

formance after using DE and $value_{orignial}$ denotes the performance before using DE. In this figure, different colors indicate that different projects are selected as source projects. It is not hard to find that using DE for optimizing parameter values of SMOTEND can improve the performance in most cases.

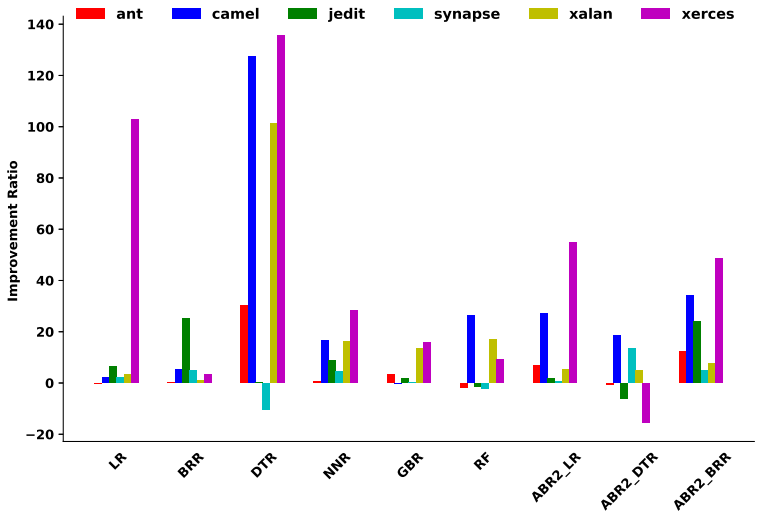
Figure 8, Figure 9 and Figure 10 use boxplot to show the performance of supervised methods not using DE and supervised methods using DE in three different scenarios by considering all the target versions. In these Figures, the method name followed by DE suffix denotes this method using DE to optimize the parameters for SMOTEND. It is not hard to find that using DE can improve the performance of SDNP whether based on FPA measure or $Kendall$ measure. From these figures, we can roughly find that LOC_D can perform significantly better than or the same as these supervised methods using DE.

Detailed comparison results can be found in Table 9, Table 10 and Table 11. These tables record the mean value of $Kendall$ and FPA for corresponding methods not using DE or using DE (the results are shown in parentheses). Then W/D/L analysis presents the number of cases, on which the corresponding method using DE performs (significantly) better than, the same as, the worse than the corresponding method not using DE. we use Wilcoxon signed-rank test to examine whether the performance difference between the corresponding method using DE and the method not using DE is statistically significant. The BH corrected p-Value is shown in rows p-value ($Kendall$) and p-value (FPA). Based on these tables, the BH corrected p-Value shows that using DE can perform significantly better than the corresponding method not using DE. Mean value and W/D/L results of $Kendall$ and FPA further verify the effectiveness of using DE to optimize the parameters of SMOTEND. These conclusions are consistent with the findings found by the previous study [11].

> In most cases, after using DE, the tuned value of parameter does not keep consistent with its default value and this optimization can significantly improve the performance of SDNP in most cases.

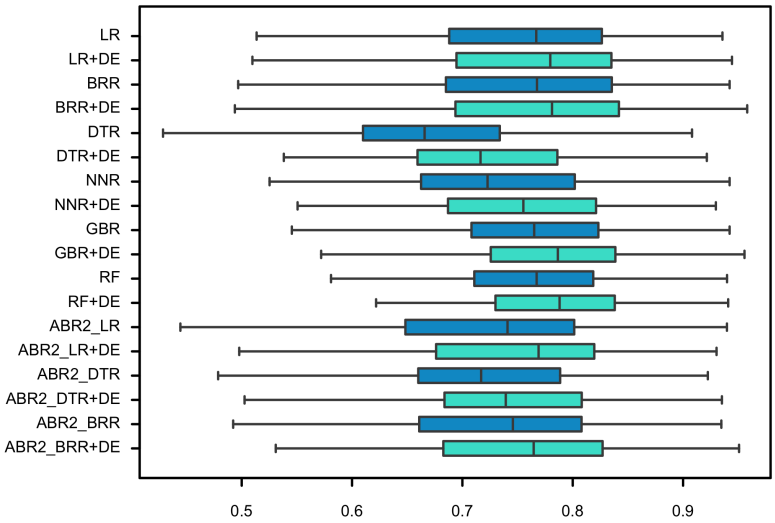


(a) *FPA*

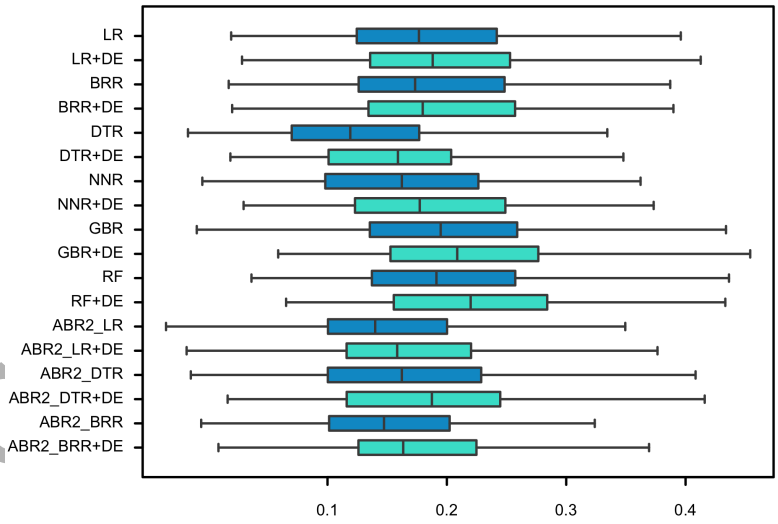


(b) *Kendall*

Figure 7: The Improvement Ratio when ivy-2.0 is set as the Target Version in the Cross-project Scenario

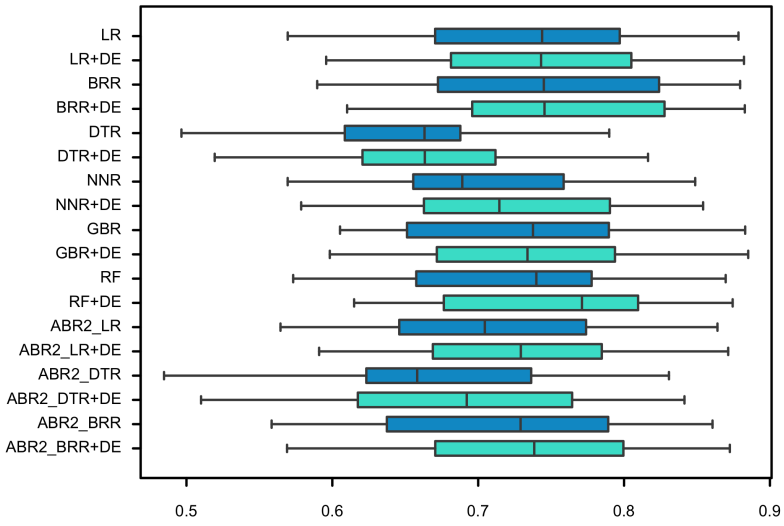


(a) *FPA*

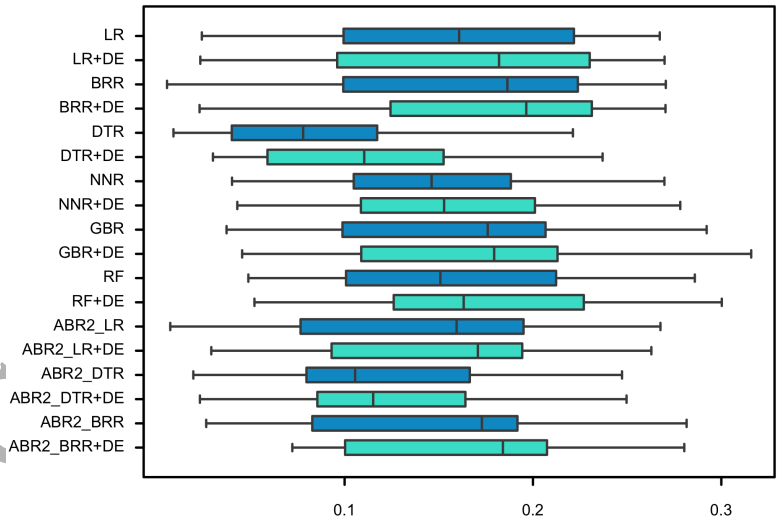


(b) *Kendall*

Figure 8: Boxplot of Supervised Methods using DE and Supervised Methods without using DE in the Within-version Scenario

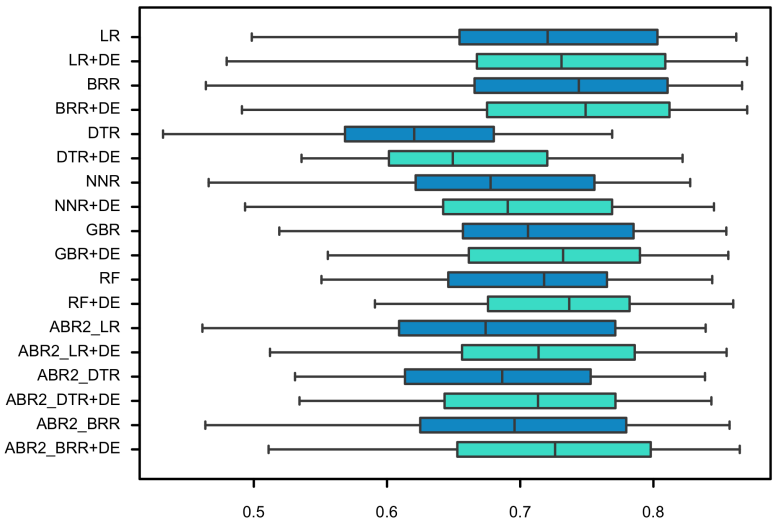


(a) *FPA*

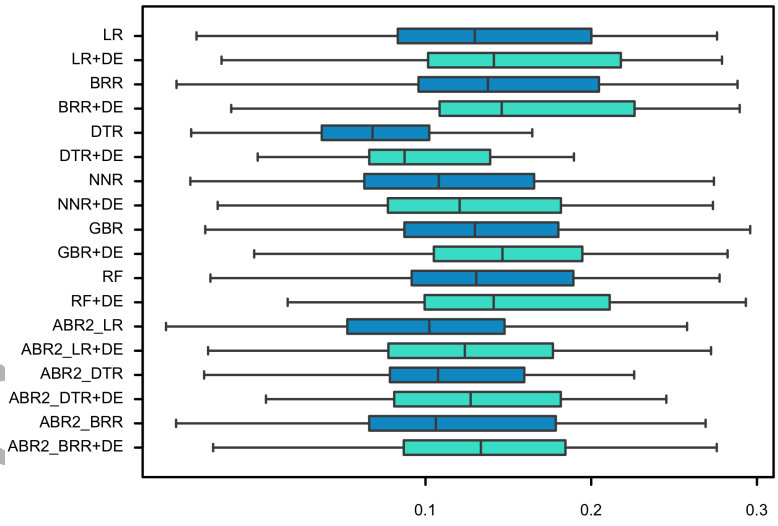


(b) *Kendall*

Figure 9: Boxplot of Supervised Methods using DE and Supervised Methods without using DE in the Cross-version Scenario



(a) *FPA*



(b) *Kendall*

Figure 10: Boxplot of Supervised Methods using DE and Supervised Methods without using DE in the Cross-project Scenario

Table 9: Comparison Results of Supervised methods using DE vs Supervised Methods not using DE in the Within-version Defect Prediction Scenario

	LR	BRR	DTR	NNR	GBR	RF	ABR2_LR	ABR2_DTR	ABR2_BRR
Kendall	0.186(**0.197**)	0.186(**0.194**)	0.126(**0.16**)	0.167(**0.186**)	0.203(**0.218**)	0.201(**0.222**)	0.15(**0.169**)	0.167(**0.186**)	0.153(**0.175**)
W/D/L (*Kendall*)	0/1/14	0/2/13	0/1/14	0/1/14	0/2/13	0/1/14	0/0/15	0/1/14	0/0/15
p value (*Kendall*)	4.31E-02	1.26E-01	2.16E-10	8.30E-04	1.99E-02	5.40E-04	5.74E-04	3.23E-03	1.05E-04
FPA	0.757(**0.768**)	0.758(**0.769**)	0.672(**0.721**)	0.729(**0.754**)	0.763(**0.781**)	0.765(**0.786**)	0.727(**0.752**)	0.721(**0.744**)	0.734(**0.755**)
W/D/L (*FPA*)	0/2/13	0/2/13	0/0/15	0/2/13	0/2/13	0/1/14	0/1/14	0/1/14	0/1/14
p value (*FPA*)	1.05E-01	1.11E-01	7.19E-13	6.69E-05	2.03E-03	1.50E-04	7.63E-04	1.53E-03	4.36E-03

Table 10: Comparison Results of Supervised methods using DE vs Supervised Methods not using DE in the Cross-version Defect Prediction Scenario

	LR	BRR	DTR	NNR	GBR	RF	ABR2_LR	ABR2_DTR	ABR2_BRR
Kendall	0.156(**0.165**)	0.165(**0.175**)	0.094(**0.118**)	0.147(**0.156**)	0.164(**0.170**)	0.159(**0.173**)	0.143(**0.153**)	0.123(**0.127**)	0.149(**0.164**)
W/D/L (*Kendall*)	4/0/11	1/1/13	0/0/15	3/0/12	6/0/9	2/1/12	4/0/11	4/0/11	3/0/12
p value (*Kendall*)	2.56E-02	3.05E-04	6.10E-05	8.54E-04	4.13E-02	1.53E-03	4.13E-02	1.07E-01	1.16E-03
FPA	0.734(**0.74**)	0.744(**0.753**)	0.656(**0.672**)	0.705(**0.719**)	0.73(**0.736**)	0.729(**0.747**)	0.709(**0.727**)	0.672(**0.689**)	0.72(**0.735**)
W/D/L (*FPA*)	5/1/9	2/1/12	3/0/12	1/0/14	8/0/7	0/0/15	1/1/13	3/0/12	1/0/14
p value (*FPA*)	1.07E-01	6.71E-03	1.25E-02	1.83E-04	4.89E-01	6.10E-05	2.01E-03	3.36E-03	4.27E-04

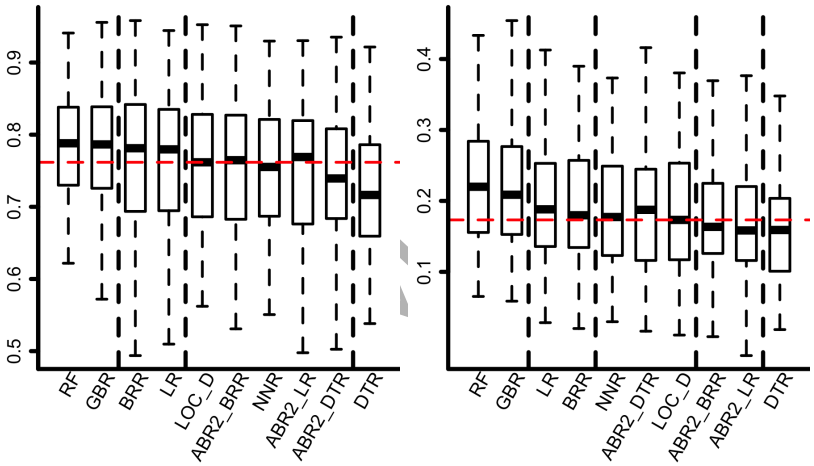
Table 11: Comparison Results of Supervised methods using DE vs Supervised Methods not using DE in the Cross-project Defect Prediction Scenario

	LR	BRR	DTR	NNR	GBR	RF	ABR2_LR	ABR2_DTR	ABR2_BRR
Kendall	0.134(**0.150**)	0.147(**0.162**)	0.072(**0.098**)	0.113(**0.127**)	0.135(**0.151**)	0.137(**0.153**)	0.108(**0.129**)	0.116(**0.129**)	0.117(**0.139**)
W/D/L (*Kendall*)	12/6/72	8/3/79	11/0/79	11/3/76	10/4/76	19/4/67	13/1/76	21/1/68	11/0/79
p value (*Kendall*)	2.04E-11	3.21E-14	1.77E-11	1.21E-13	1.42E-12	2.53E-10	1.08E-11	1.08E-08	1.68E-11
FPA	0.718(**0.732**)	0.730(**0.742**)	0.622(**0.660**)	0.682(**0.696**)	0.710(**0.726**)	0.711(**0.732**)	0.683(**0.713**)	0.685(**0.705**)	0.692(**0.721**)
W/D/L (*FPA*)	8/2/80	18/7/65	24/1/65	12/1/77	14/2/74	17/1/72	12/1/77	13/1/76	17/0/73
p value (*FPA*)	5.41E-13	1.40E-09	1.19E-08	1.27E-11	1.84E-10	1.87E-11	3.91E-13	1.60E-12	6.73E-10

5.3. Result Analysis for RQ3

RQ3: Can supervised methods using DE perform better than unsupervised methods?

The result of the Scott-Knott test between LOC_D and these 9 supervised methods using DE can be found in Figure 11, Figure 12 and Figure 13 in three different scenarios. In these figures, the horizontal red dashed lines indicate the median value of the LOC_D method, which is to help visualize the median differences between LOC_D and different supervised methods using DE.

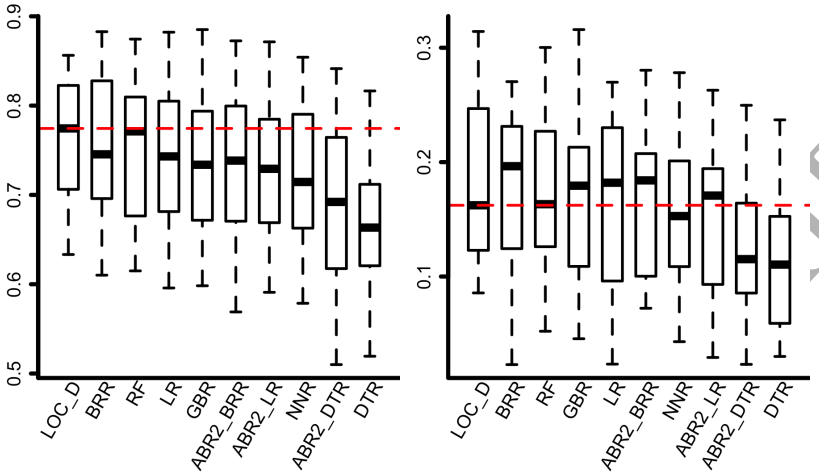


(a) *FPA* (b) *Kendall*

Figure 11: Comparison Result between Supervised Methods using DE and Unsupervised Method LOC_D in the within-version Defect Prediction Scenario based on Scott-Knott Test

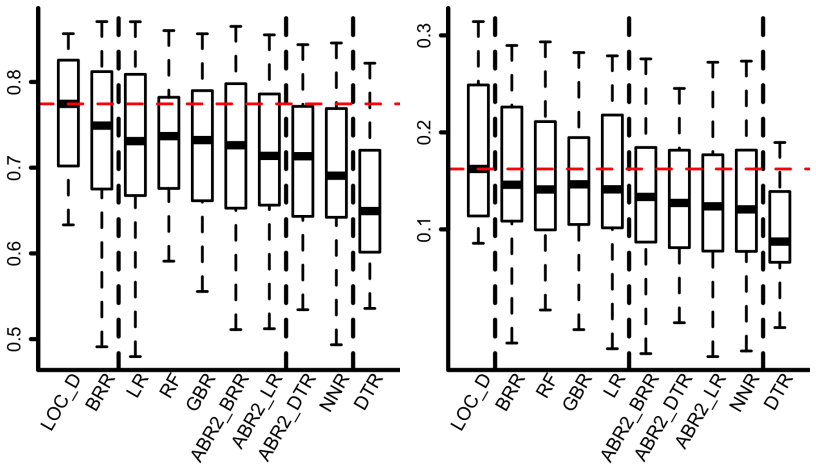
The detailed comparison results between unsupervised method LOC_D and supervised methods (including mean value, W/D/L result, # top ranks and p-Value) can be found in 12, Table 13 and Table 14 in three different scenarios.

(1)In the within-version scenario, when considering mean *Kendall* value, LOC_D performs better than 33.3% (3/9) supervised methods. The mean *Kendall* value of LOC_D is 0.184. The best method is RF and its mean *Kendall* value is 0.222. Based on W/D/L (*Kendall*) analysis, LOC_D achieves at least



(a) FPA (b) $Kendall$

Figure 12: Comparison Result between Supervised Methods using DE and Unsupervised Method LOC_D in the Cross-version Defect Prediction Scenario based on Scott-Knott Test



(a) FPA (b) $Kendall$

Figure 13: Comparison Result between Supervised Methods using DE and Unsupervised Method LOC_D in the Cross-project Defect Prediction Scenario based on Scott-Knott Test

Table 12: Results of Comparing unsupervised method LOC_D with Supervised Methods using DE in the Within-version Defect Prediction Scenario

	LR	BRR	DTR	NNR	GBR	RF	ABR2_LR	ABR2_DTR	ABR2_BRR	LOC_D
Kendall	0.197	0.194	0.160	0.186	0.218	**0.222**	0.169	0.186	0.175	0.184
W/D/L (*Kendall*)	7/6/2	7/7/1	1/4/10	3/8/4	8/7/0	10/5/0	5/4/6	5/4/6	6/3/6	
#top ranks (*Kendall*)	1	2	0	0	3	6	0	1	1	2
p value(*Kendall*)	1.72E-08	1.93E-08	4.45E-16	6.48E-01	1.40E-26	4.38E-33	1.92E-04	3.88E-01	4.60E-02	
FPA	0.768	0.769	0.721	0.754	0.781	**0.786**	0.752	0.744	0.755	0.757
W/D/L (*FPA*)	6/7/2	6/8/1	0/6/9	3/7/5	9/6/0	11/3/1	3/7/5	2/10/3	4/7/4	
#top ranks (*FPA*)	1	4	0	0	2	8	0	0	1	0
p value (*FPA*)	6.99E-08	2.71E-11	4.23E-20	4.40E-02	1.48E-20	8.08E-29	3.16E-01	3.62E-03	9.65E-01	

Table 13: Results of Comparing unsupervised method LOC_D with Supervised Methods using DE in the Cross-version Defect Prediction Scenario

	LR	BRR	DTR	NNR	GBR	RF	ABR2_LR	ABR2_DTR	ABR2_BRR	LOC_D
Kendall	0.165	0.175	0.118	0.156	0.170	0.173	0.153	0.127	0.164	**0.182**
W/D/L (*Kendall*)	7/0/8	8/0/7	3/0/12	4/0/11	7/0/8	5/0/10	4/0/11	3/0/12	6/0/9	
#top ranks (*Kendall*)	0	4	0	0	4	2	0	0	0	5
p value (*Kendall*)	2.52E-01	9.78E-01	1.53E-03	4.13E-02	3.59E-01	4.54E-01	1.81E-02	4.27E-03	1.69E-01	
FPA	0.740	0.753	0.672	0.719	0.736	0.747	0.727	0.689	0.735	**0.759**
W/D/L (*FPA*)	6/0/9	9/0/6	0/0/15	3/0/12	7/0/8	6/0/9	4/0/11	2/0/13	6/0/9	
#top ranks (*FPA*)	0	4	0	0	5	2	0	0	1	3
p value (*FPA*)	2.29E-01	8.90E-01	6.10E-05	1.81E-02	3.30E-01	4.21E-01	2.56E-02	8.54E-04	1.35E-01	

Table 14: Results of Comparing unsupervised method LOC_D with Supervised Methods using DE in the Cross-project Defect Prediction Scenario

	LR	BRR	DTR	NNR	GBR	RF	ABR2_LR	ABR2_DTR	ABR2_BRR	LOC_D
Kendall	0.150	0.162	0.098	0.127	0.151	0.153	0.129	0.129	0.139	**0.182**
W/D/L (*Kendall*)	21/1/68	34/6/50	0/0/90	0/0/90	19/0/71	12/0/78	12/0/78	4/2/84	13/3/74	
#top ranks (*Kendall*)	8	24	0	0	9	4	0	0	2	50
p value (*Kendall*)	1.12E-09	1.27E-03	1.77E-16	1.77E-16	8.67E-10	3.68E-13	2.94E-14	4.67E-16	3.58E-12	
FPA	0.732	0.742	0.660	0.696	0.726	0.732	0.713	0.705	0.721	**0.759**
W/D/L (*FPA*)	22/1/67	36/2/52	0/0/90	2/0/88	12/2/76	10/1/79	12/1/77	4/0/86	16/0/74	
#top ranks (*FPA*)	10	22	0	2	3	7	2	0	1	45
p value(*FPA*)	1.24E-07	3.72E-03	1.77E-16	2.56E-16	8.93E-13	2.73E-13	1.10E-13	3.24E-16	1.10E-12	

50% wins (or draws) when comparing to almost all the supervised methods. In most cases, LOC_D achieves at least 53.3% (8/15) wins (or draws). When counting #top ranks ($Kendall$), RF is in the first place, which can achieve top rank in 6 cases. LOC_D is in the third place, which can achieve top rank in 2 cases. When considering mean FPA value, LOC_D performs better than 55.6% (5/9) supervised methods. The mean FPA value of LOC_D is 0.757. The best method is RF and its mean FPA value is 0.786. Based on W/D/L (FPA) analysis, LOC_D achieves at least 50% wins (or draws) when comparing to almost all the supervised methods. In most cases, LOC_D achieves at least 60% (9/15) wins (or draws). When counting #top ranks (FPA), RF is in the best place, which can achieve top rank in 8 cases. Based on BH corrected p-Value and Cliffs δ, we find that based on $Kendall$, LOC_D performs significantly better than 1 supervised method (i.e., DTR) and performs significantly worse than 2 supervised methods (i.e., GBR and RF). While the remaining supervised methods do not perform significantly better than LOC_D. Based on FPA, LOC_D performs significantly better than 1 supervised methods (i.e., DTR) and performs significantly worse than 2 supervised methods (i.e., GBR and RF). While the remaining supervised methods do not perform significantly better than LOC_D.

(2) In the cross-version scenario, when considering mean $Kendall$ value, LOC_D performs better than all the supervised methods. The mean $Kendall$ value of LOC_D is 0.182. Based on W/D/L ($Kendall$) analysis, LOC_D achieves at least 50% wins (or draws) when comparing to almost all the supervised methods. In most cases, LOC_D achieves at least 60% (9/15) wins (or draws). When counting #top ranks ($Kendall$), LOC_D is in the best place, which can achieve top rank in 5 cases. BRR and GBR is in the second place, which can achieve top rank in 4 cases. When considering mean FPA value, LOC_D performs better than all the supervised methods. The mean FPA value of LOC_D is 0.759. Based on W/D/L (FPA) analysis, LOC_D achieves at least 50% wins (or draws) when comparing to almost all the supervised methods. In most cases, LOC_D achieves at least 60% (9/15) wins (or draws). When counting #top ranks (FPA), GBR is in the first place, which can achieve top

rank in 5 cases. LOC_D is in the third place, which can achieve top rank in 3 cases. Based on BH corrected p-Value and Cliffs δ, we find that whether based on $Kendall$ or FPA, LOC_D performs significantly better than 4 supervised methods (i.e., DTR, NNR, ABR2_LR, and ABR2_DTR). While the remaining supervised methods do not perform significantly better than LOC_D.

(3) In the cross-project scenario, when considering mean $Kendall$ value, LOC_D performs better than all the supervised methods. The mean $Kendall$ value of LOC_D is 0.182. Based on W/D/L ($Kendall$) analysis, LOC_D achieves at least 50% wins (or draws) when comparing to all the supervised methods. In most cases, LOC_D achieves at least 78.9% (71/90) wins (or draws). When counting #top ranks ($Kendall$), LOC_D is in the best place, which can achieve top rank in 50 cases. When considering mean FPA value, LOC_D performs better than all the supervised methods. The mean FPA value of LOC_D is 0.759. Based on W/D/L (FPA) analysis, LOC_D achieves at least 50% wins (or draws) when comparing to all the supervised methods. In most cases, LOC_D achieves at least 82.2% (74/90) wins (or draws). When counting #top ranks (FPA), both LOC_D and GBR are in the first place, which can achieve top rank in 45 cases. Based on BH corrected p-Value and Cliffs δ, we find LOC_D performs significantly better than almost all the supervised methods (except for BRR) based on whether $Kendall$ or FPA.

Even LOC_D is compared with these supervised methods using DE for optimization, previous conclusions for RQ1 still hold, especially in the cross-version scenario and cross-project scenario.

6. Discussions

6.1. Comparing with OneWay and CBS

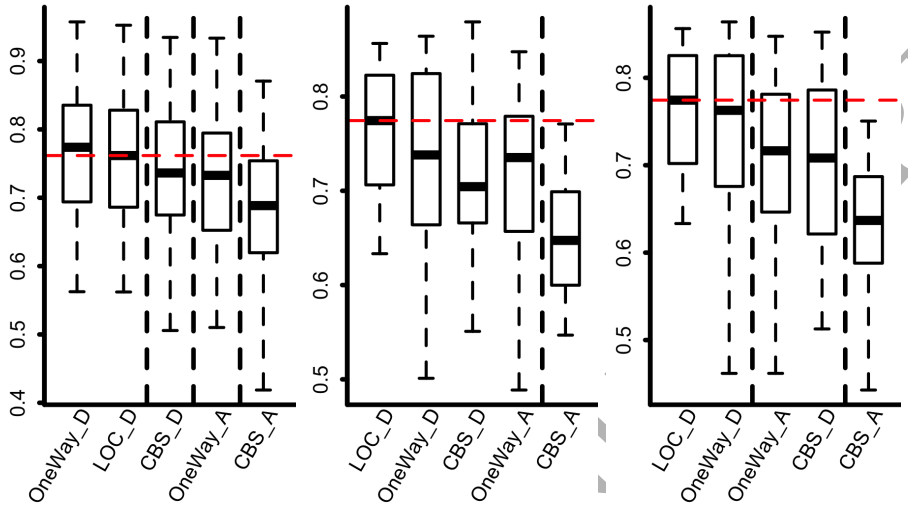
In this subsection, we want to compare LOC_D with two state-of-the-art methods (i.e., OneWay [7] and CBS [8]) proposed in recent effort-aware just-in-time defect prediction studies. Since there exist some differences between just-in-time defect prediction and software defect number prediction, we illustrate

these two methods in the context of SDNP.

OneWay method [7] is a simple supervised method. For SDNP problem, this method identifies the best method from all the unsupervised methods (introduced in Subsection 3.2) based on the analysis of the training data when considering FPA measure or $Kendall$ measure, and then applies this best unsupervised method to the testing data. Since we consider unsupervised methods with different ranking strategies, we use OneWay_A to denote OneWay method based on unsupervised methods using the ascendant ranking strategy and use OneWay_D to denote OneWay method based on unsupervised methods using the descendent strategy.

CBS method [8] is also a simple but improved supervised method. This method first builds a classifier by using Logistic regression to identify defective modules. Then it sorts the identified defective modules by a specific ranking strategy. For CBS method, we also consider SMOTEND method optimized by using differential evolutionary when building the classifier. Similar to OneWay method, we use CBS_A to denote CBS method, which uses the ascendant ranking strategy in the identified defective modules and use CBS_D to denote CBS method, which uses the descendent ranking strategy.

Final results for three different scenarios can be found in Figure 14 and Figure 15 respectively. In these figures, the horizontal red dashed lines indicate the median value of the LOC_D method, which is to help visualize the median differences between LOC_D and two state-of-the-art methods. The detailed comparison results can be found in Table 15, Table 16 and Table 17. From these figures and tables, we can find that in the within-version scenario, LOC_D can perform significantly better than CBS_A and OneWay_A. While LOC_D has the similar performance with CBS_D and OneWay_D. In the cross-version scenario, LOC_D can perform significantly better than CBS_A, CBS_D, OneWay_A. While LOC_D has the similar performance with OneWay_D. In the cross-project scenario, LOC_D can perform significantly better than all the methods (except for OneWay_D when considering FPA measure). These findings are in consistent with the conclusions by Fu and Menzies [7].

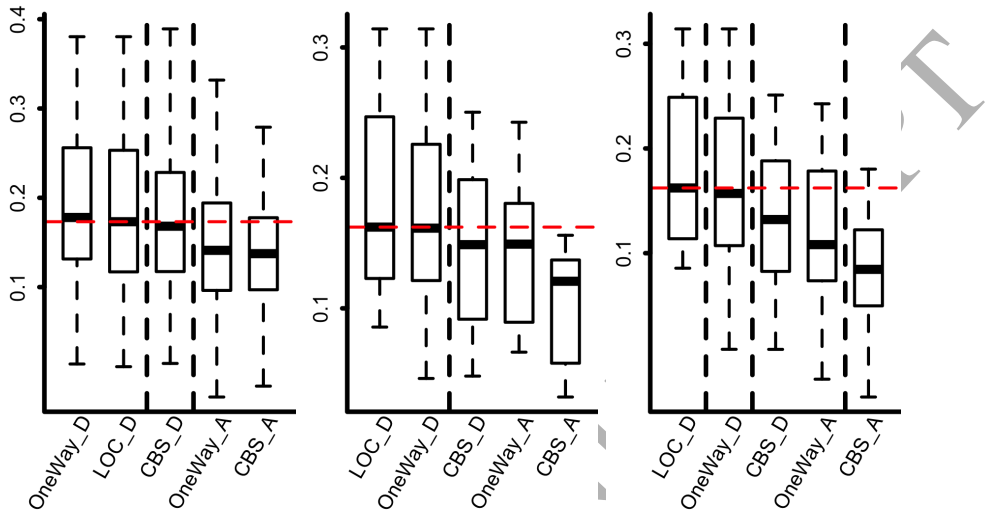


(a) Within-version Scenario (b) Cross-version Scenario (c) Cross-project Scenario

Figure 14: Comparison Result between LOC_D with OneWay and CBS methods based on Scott-Knott Test When Considering FPA measure

Table 15: Comparison Results between Unsupervised Method LOC_D with CBS and OneWay in the Within-version Scenario

	CBS_A	CBS_D	OneWay_A	OneWay_D	LOC_D
$Kendall$	0.140	0.176	0.146	**0.192**	0.184
W/D/L ($Kendall$)	0/6/9	3/5/7	0/5/10	6/7/2	
# Top Ranks ($Kendall$)	0	3	0	**7**	**7**
p value ($Kendall$)	1.43E-30	1.71E-04	8.10E-33	7.73E-09	
FPA	0.686	0.739	0.725	**0.763**	0.757
W/D/L (FPA)	0/2/13	2/6/7	0/5/10	4/9/2	
# Top Ranks (FPA)	0	2	0	7	**8**
p value (FPA)	1.46E-41	9.30E-05	5.21E-30	7.53E-06	



(a) Within-version Scenario (b) Cross-version Scenario (c) Cross-project Scenario

Figure 15: Comparison Result between LOC_D with OneWay and CBS methods based on Scott-Knott Test When Considering *Kendall* measure

Table 16: Comparison Results between Unsupervised Method LOC_D with CBS and OneWay in the Cross-version Scenario

	CBS_A	CBS_D	OneWay_A	OneWay_D	LOC_D
Kendall	0.103	0.145	0.129	0.171	**0.182**
W/D/L (*Kendall*)	0/0/15	3/0/12	1/0/14	6/4/5	
# Top Ranks (*Kendall*)	0	2	1	**8**	7
p value (*Kendall*)	6.10E-05	2.62E-03	1.83E-04	9.78E-01	
FPA	0.649	0.714	0.713	0.731	**0.759**
W/D/L (*FPA*)	0/0/15	2/0/13	2/0/13	6/3/6	
# Top Ranks (*FPA*)	0	1	1	**8**	**8**
p value (*FPA*)	6.10E-05	1.16E-03	1.16E-03	3.27E-01	

Table 17: Comparison Results between Unsupervised Method LOC_D with CBS and OneWay in the Cross-project Scenario

	CBS_A	CBS_D	OneWay_A	OneWay_D	LOC_D
Kendall	0.087	0.133	0.122	0.161	**0.182**
W/D/L (*Kendall*)	1/0/89	5/1/84	5/0/85	26/29/35	
# Top Ranks (*Kendall*)	0	5	2	52	**60**
p value (*Kendall*)	1.83E-16	1.68E-15	6.58E-16	7.45E-03	
FPA	0.636	0.703	0.705	0.739	**0.759**
W/D/L (*FPA*)	0/0/90	4/3/83	10/0/80	31/27/32	
# Top Ranks (*FPA*)	0	4	5	**51**	**51**
p value (*FPA*)	1.77E-16	1.21E-15	4.18E-14	1.16E-01	

6.2. The Usage of Unsupervised Methods

The previous study [14] showed that the majority (approximately 80%) of defects are contained in a small number (approximately 20%) of program modules. Therefore, to show the effectiveness of unsupervised methods in actual software testing, we assume that only 20% modules can be used to perform code inspection and we use *Recall*@20% to denote the ratio of detected defects. We use a simple example to illustrate *Recall*@20% measure. Suppose there are 1000 modules in the project and this project contains 20 defects. If we inspect 200 modules according to the ranked list by a specific SDNP method, we can find 10 defects. Then the value of *Recall*@20% of this SDNP method is 10/20 = 50%. The comparison results in the cross-version scenario can be found in Figure 16 and the horizontal red dashed line indicates the median value of the LOC_D method. Since supervised methods using DE can be optimized for two different measures, the supervised methods with suffix F denote the methods are optimized for *FPA* measure and the supervised methods with suffix K denote the methods are optimized for *Kendall* measure. From this figure, we can find that LOC_D method can find more defects when compared to other supervised methods. The similar conclusions can be found in other two scenarios.

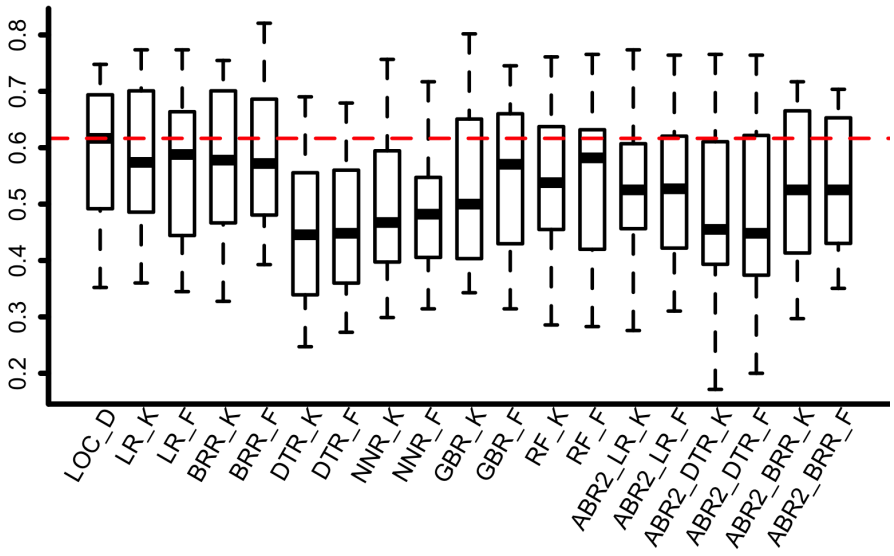


Figure 16: Comparison Results between LOC_D and Supervised Methods When Considering *Recall*@20% Measure

7. Threats to Validity

In this subsection, we mainly discuss the potential threats to validity of our empirical studies.

Threats to internal validity are mainly concerned with the uncontrolled internal factors that might have influence on the experimental results. The main internal threat is the potential faults introduced during our method implementation. To reduce this threat, we use test cases to verify the correctness of our implementation. Moreover, for regression based supervised methods (such as LR, BRR, DTR), we use the implementation of these methods supported by mature third-party library, such as packages from scikit-learn [5].

Threats to external validity are about whether the observed experimental results can be generalized to other subjects. To alleviate this threat, we consider datasets widely used by previous studies for SDNP [9][17][10][20][21].

Threats to conclusion validity are mainly concerned with inappropriate use of statistical techniques. In this paper, We use BH corrected p-Value and Cliffs δ to examine whether the performance difference between two methods are statistically significant. Moreover, Scott-Knott test is used to examine whether some methods outperform others and create a global ranking of these methods.

Threats to construct validity are about whether the performance measures used in the empirical studies reflect the real-world situation. In this paper, we mainly consider FPA and $Kendall$ rank correlation coefficient [10], which can effectively avoid the disadvantage of other performance measures, such as AAE. In the future, Spearman's rank correlation coefficient and cost effectiveness graph [52] can be further investigated.

8. Conclusion and Future Work

Software defect number prediction can be used to rank the modules and then optimize the allocation for testing resources. To the best of our knowl-

[5] http://scikit-learn.org/

edge, this is the first paper to make a comparison for these two different types of methods (i.e., supervised methods and unsupervised methods). In our empirical studies, we consider 7 real open-source projects with 24 versions in total, use FPA and $Kendall$ as our performance measures, and consider three performance evaluation scenarios (i.e., within-version scenario, cross-version scenario, and cross-project scenario). Final results show that LOC_D can perform significantly better than or the same as these supervised methods using SMOTEND. Later motivated by a recent study conducted by Agrawla and Menzies [11], we apply differential evolutionary for optimizing parameters used by SMOTEND and find that using DE can effectively improve the performance of these supervised methods too. Finally, we continue to compare LOC_D with these optimized supervised methods using DE, and previous conclusions still hold, especially in the cross-version and cross-project scenarios. Based on the above study, we suggest that researchers need to measure the modules using the LOC metric when gathering datasets and then use the unsupervised method LOC_D as the baseline method for future research on software defect number prediction problem, since this method has relatively low computation cost, is easy to implement, and has a satisfactory performance.

In the future, we want to extend our research in several ways. First we want to investigate the generalization of our empirical studies by considering more datasets from open-source projects and commercial projects. Secondly we want to investigate whether more complicated unsupervised methods (such as [53][54]) can further improve the performance. Finally, we want to resort to other novel supervised methods, such as using multi-objective optimization [55] to construct the model or using feature selection [56][57][58][26], which is used to identify and remove irrelevant features and redundant features, to further improve the performance of these supervised methods.

Acknowledgment

The authors would like to thank the editors and the anonymous reviewers for their insightful comments and suggestions, which can substantially improve the quality of this work. This work is supported in part by National Natural Science Foundation of China (Grant Nos. 61702041, 61202006), The Open Project of State Key Laboratory for Novel Software Technology at Nanjing University under (Grant Nos. KFKT2016B18, KFKT2016B12), Guangxi Key Laboratory of Trusted Software (Grant No. kx201610), The Science and Technology Project of Beijing Municipal Education Commission (Grant No. KM201811232016), and Jiangsu Government Scholarship for Overseas Studies. Xiang Chen and Dun Zhang have contributed equally for this work and they are co-first authors.

References

References

[1] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Transactions on Software Engineering 38 (6) (2012) 1276–1304.

[2] Y. Kamei, E. Shihab, Defect prediction: Accomplishments and future challenges, in: Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering, 2016, pp. 33–45.

[3] D. Radjenovic, M. Hericko, R. Torkar, A. Zivkovic, Software fault prediction metrics: A systematic literature review, Information & Software Technology 55 (8) (2013) 1397–1418.

[4] C. Tantithamthavorn, A. E. Hassan, An experience report on defect modelling in practice: Pitfalls and challenges, in: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, 2018, pp. 286–295.

[5] G. K. Rajbahadur, S. Wang, Y. Kamei, A. E. Hassan, The impact of using regression models to build defect classifiers, in: Proceedings of the International Conference on Mining Software Repositories, 2017, pp. 135–145.

[6] 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 International Symposium on Foundations of Software Engineering, 2016, pp. 157–168.

[7] W. Fu, T. Menzies, Revisiting unsupervised learning for defect prediction, in: Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2017, pp. 72–83.

[8] Q. Huang, X. Xia, D. Lo, Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction, in: Proceedings of the International Conference on Software Maintenance and Evolution, 2017, pp. 159–170.

[9] M. Yan, Y. Fang, D. Lo, X. Xia, X. Zhang, File-level defect prediction: Unsupervised vs. supervised models, in: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, 2017, pp. 344–353.

[10] X. Yu, J. Liu, Z. Yang, X. Jia, Q. Ling, S. Ye, Learning from imbalanced data for predicting the number of software defects, in: Proceedings of the International Symposium on Software Reliability Engineering, 2017, pp. 78–89.

[11] A. Agrawal, T. Menzies, Is "better data" better than "better data miners"? (on the benefits of tuning smote for defect prediction), in: Proceedings of the International Conference on Software Engineering, 2018.

[12] T. L. Graves, A. F. Karr, J. S. Marron, H. Siy, Predicting fault incidence

using software change history, IEEE Transactions on Software Engineering 26 (7) (2000) 653–661.

[13] J. Wang, H. Zhang, Predicting defect numbers based on defect state transition models, in: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, 2012, pp. 191–200.

[14] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Predicting the location and number of faults in large software systems, IEEE Transactions on Software Engineering 31 (4) (2005) 340–355.

[15] A. Janes, M. Scotto, W. Pedrycz, B. Russo, M. Stefanovic, G. Succi, Identification of defect-prone classes in telecommunication software systems using design metrics, Information Sciences 176 (24) (2006) 3711–3734.

[16] K. Gao, T. M. Khoshgoftaar, A comprehensive empirical study of count models for software fault prediction, IEEE Transactions on Reliability 56 (2) (2007) 223–236.

[17] M. Chen, Y. Ma, An empirical study on predicting defect numbers, in: Proceedings of the International Conference on Software Engineering and Knowledge Engineering, 2015, pp. 397–402.

[18] S. S. Rathore, S. Kumar, A decision tree regression based approach for the number of software faults prediction, ACM Sigsoft Software Engineering Notes 41 (1) (2016) 1–6.

[19] S. S. Rathore, S. Kumar, An empirical study of some software fault prediction techniques for the number of faults prediction, Soft Computing (2016) 1–18.

[20] S. S. Rathore, S. Kumar, Linear and non-linear heterogeneous ensemble methods to predict the number of faults in software systems, Knowledge-Based Systems 119 (2017) 232 – 256.

[21] S. S. Rathore, S. Kumar, Towards an ensemble based system for predicting the number of software faults, Expert Systems with Applications 82 (1) (2017) 357–382.

[22] H. He, E. A. Garcia, Learning from imbalanced data, IEEE Transactions on Knowledge & Data Engineering 21 (9) (2009) 1263–1284.

[23] M. Tan, L. Tan, S. Dara, C. Mayeux, Online defect prediction for imbalanced data, in: Proceedings of the International Conference on Software Engineering, 2015, pp. 99–108.

[24] S. Wang, X. Yao, Using class imbalance learning for software defect prediction, IEEE Transactions on Reliability 62 (2) (2013) 434–443.

[25] L. Chen, B. Fang, Z. Shang, Y. Tang, Tackling class overlap and imbalance problems in software defect prediction, Software Quality Journal (2016) 1–29.

[26] W. Liu, S. Liu, Q. Gu, J. Chen, X. Chen, D. Chen, Empirical studies of a two-stage data preprocessing approach for software fault prediction, IEEE Transactions on Reliability 65 (1) (2016) 38–53.

[27] M. M. Ozturk, Which type of metrics are useful to deal with class imbalance in software defect prediction?, Information & Software Technology 92 (2017) 17–29.

[28] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, Smote: synthetic minority over-sampling technique, Journal of Artificial Intelligence Research 16 (1) (2002) 321–357.

[29] 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 Transactions on Software Engineering PP (99) (2017) 1–1.

[30] 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 Transactions on Software Engineering 39 (6) (2013) 757–773.

[31] J. Liu, Y. Zhou, Y. Yang, H. Lu, B. Xu:, Code churn: A neglected metric in effort-aware just-in-time defect prediction, in: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, 2017, pp. 11–19.

[32] H. Drucker, Improving regressors using boosting techniques, in: Proceedings of the International Conference on Machine Learning, 1997, pp. 107–115.

[33] L. Torgo, P. Branco, R. P. Ribeiro, B. Pfahringer, Resampling strategies for regression, Expert Systems 32 (3) (2015) 465–476.

[34] S. Di Martino, F. Ferrucci, C. Gravino, F. Sarro, A genetic algorithm to configure support vector machines for predicting fault-prone components, in: Proceedings of the International Conference on Product Focused Software Process Improvement, 2011, pp. 247–261.

[35] W. Fu, T. Menzies, X. Shen, Tuning for software analytics: Is it really necessary?, Information & Software Technology 76 (2016) 135–146.

[36] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto, Automated parameter optimization of classification techniques for defect prediction models, in: Proceedings of the International Conference on Software Engineering, 2016, pp. 321–332.

[37] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto, The impact of automated parameter optimization on defect prediction models, IEEE Transactions on Software Engineering.

[38] W. Fu, V. Nair, T. Menzies, Why is differential evolution better than grid search for tuning defect predictors?, arXiv preprint arXiv:1609.02613.

[39] R. Storn, K. Price, Differential evolution c a simple and efficient heuristic for global optimization over continuous spaces, Journal of Global Optimization 11 (4) (1997) 341–359.

[40] A. G. Koru, K. El Emam, D. Zhang, H. Liu, D. Mathew, Theory of relative defect proneness, Empirical Software Engineering 13 (5) (2008) 473.

[41] A. G. Koru, D. Zhang, K. El Emam, H. Liu, An investigation into the functional form of the size-defect relationship for software modules, IEEE Transactions on Software Engineering 35 (2) (2009) 293–304.

[42] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener, Defect prediction from static code features: current results, limitations, new approaches, Automated Software Engineering 17 (4) (2010) 375–407.

[43] M. Jureczko, L. Madeyski, Towards identifying software project clusters with regard to defect prediction, in: Proceedings of the International Conference on Predictive Models in Software Engineering, 2010, pp. 9:1–9:10.

[44] P. He, B. Li, X. Liu, J. Chen, Y. Ma, An empirical study on software defect prediction with a simplified metric set, Information & Software Technology 59 (2015) 170–190.

[45] N. Nagappan, T. Ball, Static analysis tools as early indicators of pre-release defect density, in: Proceedings of the International Conference on Software Engineering, 2005, pp. 580–586.

[46] E. J. Weyuker, T. J. Ostrand, R. M. Bell, Comparing the effectiveness of several modeling methods for fault prediction, Empirical Software Engineering 15 (3) (2010) 277–295.

[47] M. Kendall, A new measure of rank correlation, Biometrika 30 (1-2) (1938) 81–89.

[48] B. Ghotra, S. Mcintosh, A. E. Hassan, Revisiting the impact of classification techniques on the performance of defect prediction models, in: Pro-

ceedings of the International Conference on Software Engineering, 2015, pp. 789–800.

[49] Y. Benjamini, Y. Hochberg, Controlling the false discovery rate: A practical and powerful approach to multiple testing, Journal of the Royal Statistical Society. Series B (Methodological) 57 (1) (1995) 289–300.

[50] B. Turhan, T. Menzies, A. B. Bener, J. D. Stefano, On the relative value of cross-company and within-company data for defect prediction, Empirical Software Engineering 14 (5) (2009) 540–578.

[51] S. Hosseini, B. Turhan, D. Gunarathna, A systematic literature review and meta-analysis on cross project defect prediction, IEEE Transactions on Software Engineering PP (99) (2017) 1–1.

[52] T. Jiang, L. Tan, S. Kim, Personalized defect prediction, in: Proceedings of the International Conference on Automated Software Engineering, 2014, pp. 279–289.

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

[54] J. Nam, S. Kim, Clami: Defect prediction on unlabeled datasets, in: Proceedings of the International Conference on Automated Software Engineering, 2015, pp. 452–463.

[55] X. Chen, Y. Zhao, Q. Wang, Z. Yuan, Multi: Multi-objective effort-aware just-in-time software defect prediction, Information & Software Technology 93 (2018) 1–13.

[56] C. Ni, W. S. Liu, X. Chen, Q. Gu, D. X. Chen, Q. G. Huang, A cluster based feature selection method for cross-project software defect prediction, Journal of Computer Science and Technology 32 (6) (2017) 1090–1107.

[57] I. H. Laradji, M. Alshayeb, L. Ghouti, Software defect prediction using ensemble learning on selected features, Information & Software Technology 58 (2015) 388–402.

[58] S. Liu, X. Chen, W. Liu, J. Chen, Q. Gu, D. Chen, Fecar: A feature selection framework for software defect prediction, in: Proceedings of the Annual International Computers, Software and Applications Conference, 2014, pp. 426–435.