Improving Ranking-Oriented Defect Prediction Using a Cost-Sensitive Ranking SVM

Xiao Yu, Jin Liu, Jacky Wai Keung, Qing Li, Kwabena Ebo Bennin, Zhou Xu, Junping Wang, and Xiaohui Cui

Abstract*—*Context: **Ranking-oriented defect prediction (RODP) ranks software modules to allocate limited testing resources to each module according to the predicted number of defects. Most RODP methods overlook that ranking a module with more defects incorrectly makes it difficult to successfully find all of the defects in the module due to fewer testing resources being allocated to the module, which results in much higher costs than incorrectly ranking the modules with fewer defects, and the numbers of defects in software modules are highly imbalanced in defective software datasets. Cost-sensitive learning is an effective technique in handling the cost issue and data imbalance problem for software defect prediction. However, the effectiveness of cost-sensitive learning has not been investigated in RODP models.** ***Aims:*** **In this article, we propose a cost-sensitive ranking support vector machine (SVM) (CSRankSVM) algorithm to improve the performance of RODP models.** ***Method:*** **CSRankSVM modifies the loss function of the ranking SVM algorithm by adding two penalty parameters to address both the cost issue and the data imbalance problem. Additionally, the loss function of the CSRankSVM is optimized using a genetic algorithm.** ***Results:*** **The experimental results for 11 project datasets with 41 releases show that CSRankSVM achieves 1.12%–15.68% higher average fault percentile average (FPA) values than the five existing RODP methods (i.e., decision tree regression, linear regression, Bayesian ridge regression, ranking SVM, and learning-to-rank (LTR)) and 1.08%–15.74% higher average FPA values than the four data imbalance learning methods (i.e., random undersampling and a synthetic minority oversampling technique; two data resampling methods; RankBoost, an ensemble learning method; IRSVM, a CSRankSVM method for information retrieval).** ***Conclusion:*** **CSRankSVM is capable of handling the cost issue and data imbalance problem in RODP methods and achieves better performance. Therefore, CSRankSVM is recommended as an effective method for RODP.**

***Index Terms*—Cost-sensitive learning, data imbalance, ranking-oriented defect prediction (RODP).**

Manuscript received June 1, 2018; revised October 20, 2018 and April 21, 2019; accepted July 14, 2019. Date of publication August 22, 2019; date of current version March 2, 2020. This work was supported in part by the National Key R&D Program of China under Grant 2018YFC1604000, in part by the National Natural Science Foundation of China under Grant 61572374, Grant U163620068, Grant U1135005, Grant 61572371, and Grant 61772525, in part by the Open Fund of Key Laboratory of Network Assessment Technology from CAS, Guangxi Key Laboratory of Trusted Software under Grant kx201607, in part by the Academic Team Building Plan for Young Scholars from Wuhan University under Grant WHU2016012, in part by the General Research Fund of the Research Grants Council of Hong Kong under Grant 11208017, in part by the research funds of City University of Hong Kong under Grant 9678149, Grant 7005028, and Grant 9678149, in part by the Research Support Fund by Intel under Grant 9220097, and in part by a start-up fund under Grant 9B0V from the Hong Kong Polytechnic University. Associate Editor: NASAC Special. *(Corresponding authors: Jin Liu; Jacky Wai Keung.)*

X. Yu is with the School of Computer Science, Wuhan University, Wuhan 430072, China, and also with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: xiaoyu_whu@yahoo.com).

J. Liu is with the School of Computer Science, Wuhan University, Wuhan 430072, China, with the Key Laboratory of Network Technology, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100000, China, and also with the Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541000, China (e-mail: jinliu@whu.edu.cn).

J. W. Keung is with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: jacky.keung@cityu.edu.hk).

Q. Li is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong (e-mail: csqli@comp.polyu.edu.hk).

K. E. Bennin is with the Department of Software Engineering, Blekinge Institute of Technology, 37134 Karlskrona, Sweden (e-mail: kwabena.ebo.bennin@bth.se).

Z. Xu is with the School of Computer Science, Wuhan University, Wuhan 430072, China (e-mail: zhouxullx@whu.edu.cn).

J. Wang is with the Laboratory of Precision Sensing and Control Center, Institute of Automation, Chinese Academy of Sciences, Beijing 100000, China (e-mail: wangjunping@bupt.edu.cn).

X. Cui is with the School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China (e-mail: xcui@whu.edu.cn).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2019.2931559

NOMENCLATURE

A. Acronyms and Abbreviations

SDP	Software defect prediction.
SLOC	Source lines of code.
RODP	Ranking-oriented defect prediction.
RMSE	Root-mean-square error.
AAE	Average absolute error.
CSRankSVM	Cost-sensitive ranking support vector machine (SVM).
LR	Linear regression.
FPA	Fault percentile average.
RUS	Random undersampling.
SMOTE	Synthetic minority oversampling technique.
PR	Poisson regression.
GP	Genetic programming.
DTR	Decision tree regression.
BRR	Bayesian ridge regression.
CE	Cost effective.
CLC	Cumulative lift chart.

B. Notations

S	Defect dataset, which can be written as $\{M_i = (\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$.
M_i	Software module, which can be written as $(\boldsymbol{x}_i, y_i)$.
$\boldsymbol{x}_i$	Feature vector of M_i, which can be written as $(x_1, x_2, \ldots, x_d)$.

x_i	Value of the *i*th feature.
y_i	Number of defects in M_i.
n	Number of modules in S.
f	Linear function.
$\boldsymbol{w}$	Vector of weights, which can be written as $(w_1, w_2, \ldots, w_d)$.
w_i	Value of the *i*th weight.
S'	Module pair dataset, which can be written as $\{P_i = (\boldsymbol{p}_i, r_i)\}_{i=1}^{m}$
P_i	Module pairs, which can be written as $(\boldsymbol{p}_i, r_i)$.
$\boldsymbol{p}_i$	Feature vector of P_i.
r_i	Label of P_i.
m	Number of module pairs in S'.
$[x]_+$	Function max (x, 0).
C	Constant.
ξ_i	Slack variable.

I. Introduction

SOFTWARE defect prediction (SDP) is one of the most active research fields in software engineering [1], [2]. It aims to predict whether a particular software module is defective based on certain software features, such as source lines of code (SLOC) and change information. More testing resources can be allocated to modules that are likely to be defective [3]–[5]. In recent years, researchers have proposed many SDP approaches [6]–[10] using binary classification algorithms. However, these approaches are restricted to predict whether a new module is defective. Conventional classification models thus offer less benefit in crucial scenarios in which software quality teams want to focus more attention or scarce resources on the most defective modules (i.e., modules with the most bugs) among all likely defective modules [7], [11]. Ranking-oriented defect prediction (RODP) ranks software modules according to their predicted number of defects and guides software testers to focus effort on the modules with more defects. Therefore, RODP can aid the allocation of limited testing resources more efficiently than SDP. As an illustration, we present a simple example in Fig. 1.

Example 1: Suppose that a new software project with 100 software modules is to be tested and the testing team can only test a small portion of these modules (e.g., 20 modules) due to a deadline. They first use historical defect datasets (including software features and the number of defects) to build the usual classification model to predict whether these modules are defective or a ranking-oriented model to rank these modules based on their number of defects. They then extract the same software features of the 100 software modules. Finally, they use the learned models to predict or rank the defect proneness of these modules. Given that 30 modules are predicted to be defective by the classification model, the test team faces the challenge of determining which of the 20 modules among the 30 predicted defective modules should be tested. However, the test team can focus on and test the first 20 modules according to the results of the ranking-oriented model, which ranks the modules according to their number of defects. Therefore, ranking software modules can be more useful than predicting whether a module is defective when testing resources are limited. As this approach is based on learning to rank techniques, we call it RODP.

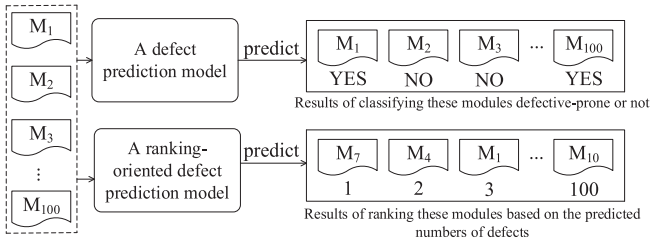

Fig. 1. Difference between the traditional SDP model and an RODP model.

Two approaches can be used to construct RODP models: regression techniques and learning to rank techniques. Regression techniques first try to predict the number of defects in software modules using regression algorithms and then use the numbers to rank the modules. These regression-based approaches construct RODP models by minimizing the total differences in the predicted and actual numbers of defects. However, predicting the precise number of defects in a module is difficult due to insufficient high-quality historical data [5], although it is more useful than predicting the order of modules. In addition, regression-based approaches with higher predictive accuracy (the average absolute error [AAE] or smaller root-mean-square error [RMSE] value) may result in a worse ranking. For instance, assuming that there are three software modules, M_1, M_2, and M_3, which contain 5, 4, and 3 defects, respectively, model P predicts that M_1, M_2, and M_3 contain 5, 2, and 3 defects, respectively, whereas model Q predicts that M_1, M_2, and M_3 contain 4, 3, and 2 defects, respectively. Although model P results in a lower AAE or RMSE value, the correct ranking of model Q is exactly what we desire for RODP.

In practice, RODP mainly aims to predict which modules are likely to contain more defects. Learning to rank software modules directly should be a much more natural RODP method than ranking the modules according to the predicted precise number of defects using regression algorithms [5]. Nguyen *et al.* [12] investigated two learning to rank algorithms (i.e., ranking SVM [13] and RankBoost [14]) for RODP and found that ranking SVM outperforms the linear regression (LR) algorithm 4%–21% in terms of the Spearman rank correlation coefficient. The superior performance of ranking SVM over the LR algorithm may be attributable to the difference in the training utility function. Ranking SVM transforms RODP into a problem of classifying module pairs into two classes (i.e., correctly ranked and incorrectly ranked). Here, an incorrect ranking means that a module with fewer defects is ranked ahead of a module with more defects in a given module pair. The goal of ranking SVM is to minimize the number of incorrectly ranked module pairs, whereas the LR algorithm aims to minimize the total differences in the predicted and actual numbers of defects. As the final prediction, performance is measured based on the rank similarity (i.e., Spearman rank correlation coefficient) rather than the total prediction errors (i.e., RMSE and AAE), the training procedure of ranking SVM is better optimized for that measure.

However, the following two key issues must be considered when applying ranking SVM to RODP.

1) In practice, it is more important to rank the modules with more defects correctly so that testers can focus their effort

TABLE I
COST MATRIX

	Predict defective	Predict nondefective
Actual defective	0	$C(1,-1)$
Actual non-defective	$C(-1, 1)$	0

on the software modules that rank first. Ranking a module with more defects incorrectly implies that fewer testing resources are allocated to the module, which makes it difficult to successfully find and fix all of the defects in the module. As failure to find a defect can severely degrade software quality, ranking a module with more defects incorrectly incurs a much higher cost than incorrectly ranking the modules with fewer defects. However, ranking SVM considers the costs between ranking the modules with more defects incorrectly and ranking the modules with fewer defects incorrectly the same way.

2) Ranking SVM is a pairwise learning to rank algorithm, which focuses on accurately predicting the relative order between the two modules. Therefore, ranking SVM must first transforms the modules in a defect dataset into module pairs. Then, a classification model is trained based on these module pairs. Finally, ranking SVM utilizes the classification model to predict which module contains more defects in a module pair. However, after transforming the modules in an imbalanced defect dataset into module pairs, these module pairs are also imbalanced. For example, the modules with more than four defects in the Ant project (see Table I) occupy only a small part of it, but the nondefective modules occupy a great part of this project, followed by the modules with one defect. That is, there are a few module pairs that are composed of modules with many defects and others, but many module pairs that are composed of modules with one defect and others. When ranking SVM is trained on these imbalanced module pairs, the trained model is biased toward the module pairs that are composed of the modules with one defect and others. That is, the trained model can rank the modules with one defect more accurately than it can rank the modules with more defects.

Cost-sensitive learning is an effective technique in handling the cost and data imbalance problem for SDP. For example, Liu *et al.* [41] proposed a two-stage cost-sensitive learning method for SDP, which incorporates cost information in both the classification stage and the feature selection stage. Jing *et al.* [42] proposed a cost-sensitive discriminative dictionary learning method for SDP. The experimental results showed that these cost-sensitive learning methods could improve the performance of the SDP models. However, the effectiveness of cost-sensitive learning has not been investigated on the performance RODP models.

Accordingly, we propose a CSRankSVM algorithm for RODP. Specifically, the CSRankSVM modifies the loss function of ranking SVM. To address the cost issue, the CSRankSVM increases the penalty when a module with more defects is ranked incorrectly, such that the modules with more defects can be ranked correctly. To solve the data imbalance problem, the CSRankSVM places more weight on module pairs that are composed of modules with many defects and others such that the training can be conducted equally over all module pairs. Then, the modified cost function of the CSRankSVM is optimized using the genetic algorithm.

We evaluate the CSRankSVM against the five existing methods using 11 project datasets with 41 releases. The experimental results show that the CSRankSVM achieves the highest average FPA value of 0.723 and outperforms the five methods with more than half of the datasets. We also compare the CSRankSVM with the three data imbalance learning methods to investigate the effectiveness of the cost-sensitive strategies of the CSRankSVM. The experimental results show that the CSRankSVM improves the average FPA value of RUS by at least 1.08% and at most 15.21%, of the SMOTE by at least 1.90% and at most 15.74%, of an ensemble learning method (RankBoost) by 3.73%, and of a cost-sensitive learning ranking SVM method for information retrieval (IRSVM) by 9.01%.

The main contributions of this article are summarized as follows.

1) We propose a CSRankSVM algorithm for RODP. This is the first attempt to introduce cost-sensitive learning to address the cost issue and the data imbalance problem for RODP models.
2) We conduct a comprehensive experiment to compare the effectiveness of the CSRankSVM with that of the five existing RODP models and five data imbalance learning methods on 11 real-world project datasets with 41 releases. Our extensive experiments show that the CSRankSVM achieves encouraging results compared with these baseline methods.

The rest of this article is organized as follows. Section II presents the related work. Section III introduces the ranking SVM algorithm. Section IV proposes our CSRankSVM method for RODP. Section V details the experimental setup. Section VI provides the experimental results. Section VII discusses the potential threats to validity. Finally, Section VIII concludes this article.

II. RELATED WORK

In this section, we review the existing SDP methods and RODP methods.

A. SDP Methods

In recent years, researchers have proposed a number of SDP methods using classification techniques, such as neural networks [15]–[17], support vector machines [18]–[20], decision trees [21], [22], and Bayesian methods [23]–[27]. Based on various software features, these methods learn from historical defect datasets to predict whether a particular software module is defective. However, defect datasets usually contain many more nondefective modules than defective ones. Therefore, the SDP models are generally biased toward nondefective modules [28], which degrade the prediction performance. Therefore, a number of data imbalance learning approaches have been proposed, such as data sampling [28]–[34], ensemble learning [35]–[40], and cost-sensitive learning [41]–[45].

Data sampling techniques include oversampling and undersampling. Oversampling adds synthetic defective modules to

achieve the class-balanced state, whereas undersampling removes nondefective modules. One of the main drawbacks of data sampling techniques is that adding and deleting modules increases the number of false alarms [28], and the loss of original information [42], respectively. Ensemble learning is another data imbalance learning approach, which combines multiple individual SDP models to generate a stronger SDP model (regardless of whether defect datasets are imbalanced). However, how to effectively utilize and guarantee the diversity of the individual SDP models has not been addressed efficiently [45].

In addition to the aforementioned data sampling and ensemble learning approaches, the cost-sensitive learning techniques are another major category of approaches used to solve the data imbalance problem. For the two-class SDP problem, the cost matrix is given in Table I. In Table I, $C(1, -1)$ represents the cost of misclassifying a defective module as nondefective, whereas $C(-1, 1)$ represents the cost of misclassifying a nondefective module as defective. As the failure to find a defect can severely degrade software quality, misclassifying a defective module incurs a higher cost than misclassifying a nondefective module. Therefore, cost-sensitive learning methods aim to generate SDP models with minimal misclassification costs. In recent years, some cost-sensitive learning approaches for SDP have been proposed. For example, Liu *et al.* [41] proposed a two-stage cost-sensitive learning method for SDP that incorporates cost information in both the classification stage and the feature selection stage. Jing *et al.* [42] proposed a cost-sensitive discriminative dictionary learning method for SDP. We propose a cost-sensitive learning method for RODP. However, the proposed RODP method is different from existing approaches in which it assigns varying costs to incorrectly ranking module pairs and increases the penalty when a module with more defects is ranked incorrectly.

B. RODP Methods

Two approaches can be used to construct the RODP models: regression techniques and learning to rank techniques [5]. Regression techniques first try to predict the number of defects in software modules using regression algorithms and then use such numbers to rank them. In recent years, a number of regression algorithms have been applied to predict the number of defects, such as Poisson regression [46]–[49] and genetic programming [50]–[52], and DTR [53]. In addition, Chen and Ma [54] and Rathore and Kumar [55] investigated regression algorithms for predicting the number of defects and found that DTR, LR, and BRR performed better in terms of the RMSE and AAE. However, these regression-based approaches construct the RODP models by minimizing the total differences in the predicted and actual numbers of defects instead of optimizing the ranking performance measures. This may be problematic, as a good model with higher predictive accuracy (i.e., a smaller RMSE or AAE value) may conduct poorer module ranking [5].

Compared with regression-based approaches, studies using learning to rank techniques for RODP have been very limited. Nguyen *et al.* [12] investigated two pairwise learning to rank algorithms (i.e., ranking SVM and RankBoost) for RODP and found that these algorithms outperform the LR algorithm 4%–21% in terms of the Spearman rank correlation coefficient. Yang *et al.* [5] proposed a learning to rank approach for RODP by directly optimizing the ranking performance (i.e., FPA). However, these approaches do not consider the cost and the data imbalance problems.

III. Preliminaries

In this section, we first introduce the ranking SVM algorithm. We then analyze two important issues when adapting ranking SVM to RODP.

A. Ranking SVM

A software module can be written as $M_i = (\boldsymbol{x}_i, y_i)$, where $\boldsymbol{x}_i = (x_1, x_2, \ldots, x_d)$ is a d-dimensional software feature vector of the module and y_i is the number of defects in the module. A software dataset can be represented as $S = \{M_i = (\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$, where n is the number of modules in S. The goal of RODP is to learn from S to obtain a ranking function f to rank two modules with a right preference relation

$$M_j \succ M_k \Leftrightarrow f(\boldsymbol{x}_j) > f(\boldsymbol{x}_k) \tag{1}$$

where $M_j \succ M_k$ indicates that the number of defects in M_j is larger than the number of defects in M_k and f is a ranking function.

Herbrich *et al.* [13] proposed the ranking SVM algorithm, which transforms the abovementioned learning task into that of learning a binary classifier on module pairs. First, ranking SVM assumes that the ranking function f is a linear function

$$f(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle \tag{2}$$

where $\boldsymbol{w} = (w_1, w_2, \ldots, w_{\mathrm{d}})$ is a vector of weights and $\langle \cdot, \cdot \rangle$ is an inner product. When (2) is inserted into (1), we obtain the following:

$$M_j \succ M_k \Leftrightarrow \langle \boldsymbol{w}, \boldsymbol{x}_j - \boldsymbol{x}_k \rangle > 0. \tag{3}$$

The relation $M_j \succ M_k$ of the module pair (M_j, M_k) is then expressed by a new vector $\boldsymbol{x}_j - \boldsymbol{x}_k$. Next, ranking SVM takes any two modules whose numbers of defects are different in S and their relation to create a module pair dataset $S' = \{P_i = (\boldsymbol{p}_i, r_i)_{i=1}^{m}\}$, where $\boldsymbol{p}_i = \boldsymbol{x}_j - \boldsymbol{x}_k$ is the vector of the module pair P_i, r_i is the label of the module pair P_i, and m is the number of module pairs in S'. If the number of defects in X_j is larger than the number of defects in X_k, $r_i = 1$. Otherwise, $r_i = -1$. Finally, an SVM model is constructed using the training dataset S'.

Constructing an SVM model is formalized as the following quadratic optimization problem [56]:

$$\min_{w,\xi} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{\mathrm{m}} \xi_i \tag{4}$$

subject to $\xi_i \geq 0$, $\boldsymbol{r}_i\langle \boldsymbol{w}, \boldsymbol{p}_{\mathrm{i}} \rangle \geq 1 - \xi_i$, $i = 1, 2, \ldots, m$, where C is a constant and ξ_i is a slack variable.

Note that the optimization in (4) is equivalent to the following nonconstrained optimization problem (i.e., the minimization of

the regularized hinge loss function) [56]:

$$\min_{\boldsymbol{w}} \sum_{\mathrm{i}=1}^{\mathrm{m}} \left[1 - \mathrm{r_i} \left\langle \boldsymbol{w}, \boldsymbol{p_i} \right\rangle\right]_+ + \lambda \|\boldsymbol{w}\|^2 \tag{5}$$

where $[x]_+$ denotes the function max (x, 0) and $\lambda = \frac{1}{2}$.

Example 2: Suppose that there are ten software modules (i.e., M_0, M_1, …, M_9), where M_0 has five defects, M_1, M_2, and M_3 have one defect, and the others are nondefective.

For example, M_0 and M_1 are two modules that can generate two module pairs, specifically $(\boldsymbol{x}_0 - \boldsymbol{x}_1, 1)$ and $(\boldsymbol{x}_1 - \boldsymbol{x}_0, -1)$. As $[1 - (1 \times \langle \boldsymbol{w}, \boldsymbol{x}_0 - \boldsymbol{x}_1 \rangle)]_+$ is equal to $[1 - (-1 \times \langle \boldsymbol{w}, x_1 - \boldsymbol{x}_0 \rangle)]_+$, $(\boldsymbol{x}_1 - \boldsymbol{x}_0, -1)$ is redundant. Therefore, we discard the negative module pairs in the learning of ranking SVM and our method, as they are redundant. We can use the ten modules to generate a module pair dataset S' = {$(\boldsymbol{x}_0 - \boldsymbol{x}_1, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_2, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_3, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_4, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_5, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_6, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_7, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_8, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_9, 1)$, $(\boldsymbol{x}_1 - \boldsymbol{x}_4, 1)$, $(\boldsymbol{x}_1 - \boldsymbol{x}_5, 1)$, $(\boldsymbol{x}_1 - \boldsymbol{x}_6, 1)$, $(\boldsymbol{x}_1 - \boldsymbol{x}_7, 1)$, $(\boldsymbol{x}_1 - \boldsymbol{x}_8, 1)$, $(\boldsymbol{x}_1 - \boldsymbol{x}_9, 1)$, $(\boldsymbol{x}_2 - \boldsymbol{x}_4, 1)$, $(\boldsymbol{x}_2 - \boldsymbol{x}_5, 1)$, $(\boldsymbol{x}_2 - \boldsymbol{x}_6, 1)$, $(\boldsymbol{x}_2 - \boldsymbol{x}_7, 1)$, $(\boldsymbol{x}_2 - \boldsymbol{x}_8, 1)$, $(\boldsymbol{x}_2 - \boldsymbol{x}_9, 1)$, $(\boldsymbol{x}_3 - \boldsymbol{x}_4, 1)$, $(\boldsymbol{x}_3 - \boldsymbol{x}_5, 1)$, $(\boldsymbol{x}_3 - \boldsymbol{x}_6, 1)$, $(\boldsymbol{x}_3 - \boldsymbol{x}_7, 1)$, $(\boldsymbol{x}_3 - \boldsymbol{x}_8, 1)$, $(\boldsymbol{x}_3 - \boldsymbol{x}_9, 1)$}, where S' only contains the positive module pairs.

B. Cost and Data Imbalance Analysis

To adapt ranking SVM to RODP, the following two important issues must be considered.

1) The correct ranking of the ten modules in Example 2 is given and two other rankings are required to be made.

Correct ranking: M_0, M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, M_9

Ranking 1: M_1, M_0, M_2, M_3, M_4, M_5, M_6, M_7, M_8, M_9

Ranking 2: M_0, M_1, M_2, M_4, M_3, M_5, M_6, M_7, M_8, M_9

Both Rankings 1 and 2 are incorrect, as Ranking 1 incorrectly ranks M_0 and M_1 and Ranking 2 incorrectly ranks M_3 and M_4. In practice, the cost of the incorrect ranking in Ranking 1 is higher than that of the incorrect ranking in Ranking 2, as it is crucial to rank the modules with more defects accurately, so that the test team can focus their effort on the software modules that list first. That is, a larger penalty should be added to Ranking 1. However, ranking SVM does not take this into consideration and assumes that the costs of the incorrect rankings in Rankings 1 and 2 are equal.

2) *Definition 1:* A positive module pair $P = (M_p, M_q)$ can be labeled with a rank label $R_{j,k}$, where j is the number of defects in M_p and k is the number of defects in M_q.

According to Definition 1, we can transform the loss function in (5) into the following formula:

$$\min_{\boldsymbol{w}} \underbrace{\sum\nolimits_{i=1}^{m_{j,k}} \left[1 - r_i \left\langle \boldsymbol{w}, \boldsymbol{p}_i \right\rangle\right]_+}_{R_{j,k}} + , \ldots , + \underbrace{\sum\nolimits_{i=1}^{m_{1,0}} \left[1 - r_i \left\langle \boldsymbol{w}, \boldsymbol{p}_i \right\rangle\right]_+}_{R_{1,0}} + \lambda \|\boldsymbol{w}\|^2 \tag{6}$$

where $m_{j,k}$ is the number of the module pairs of rank $R_{j,k}$.

Example 3: Transforming the ten modules in Example 2 into module pairs and labeling them with Definition 1 yields three (=1 × 3) module pairs of *Rank*$_{5,1}$ [i.e., $(\boldsymbol{x}_0 - \boldsymbol{x}_1, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_2, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_3, 1)$], 6 (=1 × 6) module pairs of *Rank*$_{5,0}$ [i.e., $(\boldsymbol{x}_0 - \boldsymbol{x}_4, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_5, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_6, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_7, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_8, 1)$, $(\boldsymbol{x}_0 - \boldsymbol{x}_9, 1)$], and 18 (=3 × 6) module pairs of *Rank*$_{1,0}$ [i.e., $(\boldsymbol{x}_1 - \boldsymbol{x}_4, 1)$, $(\boldsymbol{x}_1 - \boldsymbol{x}_5, 1)$, $(\boldsymbol{x}_1 - \boldsymbol{x}_6, 1)$, $(\boldsymbol{x}_1 - \boldsymbol{x}_7, 1)$, $(\boldsymbol{x}_1 - \boldsymbol{x}_8, 1)$, $(\boldsymbol{x}_1 - \boldsymbol{x}_9, 1)$, $(\boldsymbol{x}_2 - \boldsymbol{x}_4, 1)$, $(\boldsymbol{x}_2 - \boldsymbol{x}_5, 1)$, $(\boldsymbol{x}_2 - \boldsymbol{x}_6, 1)$, $(\boldsymbol{x}_2 - \boldsymbol{x}_7, 1)$, $(\boldsymbol{x}_2 - \boldsymbol{x}_8, 1)$, $(\boldsymbol{x}_2 - \boldsymbol{x}_9, 1)$, $(\boldsymbol{x}_3 - \boldsymbol{x}_4, 1)$, $(\boldsymbol{x}_3 - \boldsymbol{x}_5, 1)$, $(\boldsymbol{x}_3 - \boldsymbol{x}_6, 1)$, $(\boldsymbol{x}_3 - \boldsymbol{x}_7, 1)$, $(\boldsymbol{x}_3 - \boldsymbol{x}_8, 1)$, $(\boldsymbol{x}_3 - \boldsymbol{x}_9, 1)$].

That is, the module pairs are also imbalanced. In this case, the loss function of the module pairs of *Rank*$_{5,1}$ and *Rank*$_{5,0}$ occupies a very small proportion of the total loss function. The optimal result of (6) would bias the module pairs of *Rank*$_{1,0}$ that occupy the larger proportion of the total loss function. The minimum prediction error cannot be attained on the module pairs of *Rank*$_{5,1}$ and *Rank*$_{5,0}$. This issue is also not reflected in the ranking SVM algorithm.

IV. CSRankSVM for RODP

In this section, we propose a CSRankSVM algorithm for RODP to address the two issues mentioned above.

A. Loss Function

The strategy of cost-sensitive learning is to modify the cost of the module pairs of different ranks. Specifically, we add a penalty parameter $\mu_{j,k}$ to increase the penalty when a module with more defects is ranked incorrectly and we add a penalty parameter $\eta_{j,k}$ to place higher weight on the module pairs that occupy a very small proportion of the total loss function. As a result, the loss function in (6) is transformed into the following formula:

$$L(\boldsymbol{w}) = \underbrace{\sum\nolimits_{i=1}^{m_{j,k}} \mu_{j,k}\eta_{j,k}\left[1 - r_i \left\langle \boldsymbol{w}, \boldsymbol{p}_i \right\rangle\right]_+}_{R_{j,k}} + , \ldots , + \underbrace{\sum\nolimits_{i=1}^{m_{1,0}} \mu_{1,0}\eta_{1,0}\left[1 - r_i \left\langle \boldsymbol{w}, \boldsymbol{p}_i \right\rangle\right]_+}_{R_{1,0}} + \lambda \|\boldsymbol{w}\|^2. \tag{7}$$

A key issue with cost-sensitive learning is the definition of the penalty parameters. We use a heuristic method to estimate the value of $\mu_{j,k}$ (see Algorithm 1). We first create a correct ranking for all of the modules in the defective software dataset S and calculate the ranking performance of this correct ranking (Lines 1 and 2). We use the FPA as the ranking performance measure (see Section V-B). In principle, any other ranking performance measures can be used. The FPA value of this correct ranking is denoted by $\mathrm{FPA_{correct}}$. For each module pair $P = (M_p, M_q)$ of *Rank* R_{jk}, we swap the two modules in P (Lines 5 and 6). This way, we obtain a new ranking and we can calculate the FPA value for it, denoted by FPA_{pq} (Line 7). There is often a decrease between $\mathrm{FPA_{correct}}$ and FPA_{pq}. Finally, we calculate the average drop over all module pairs of *Rank* R_{jk} and take it as the value of μ_{jk} (Line 11). With such a parameter, we can increase the penalty when a module with more defects is ranked incorrectly.

Example 4: Considering the modules in Example 2, the correct ranking for all of the modules is M_0, M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, M_9 and the FPA value of the correct ranking

Algorithm 1: Calculation of $\mu_{j,k}$.

Input: Software dataset $S = \{M_i = (\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$
Output: $\mu_{j,k}$
1: Create a correct ranking for all of the modules in *S*;
2: Obtain the FPA value ($\text{FPA}_{\text{correct}}$) of this correct ranking;
3: Drop = 0;
4: Count = 0;
5: **for** each module pair $P = (M_p, M_q)$ of *Rank* R_{jk}:
6: Swap M_p and M_q, and obtain a new ranking;
7: Calculate the FPA value (FPA_{pq}) for it;
8: Drop = Drop+ $\text{FPA}_{\text{correct}}$ - FPA_{pq};
9: Count++;
10: end for
11: $\mu_{j,k}$ = Drop/Count;
12: **return** $\mu_{j,k}$;

Algorithm 2: Calculation of $\eta_{\text{j,k}}$.

Input: Module pairs dataset $S' = \{P_i = (\boldsymbol{p}_i, r_i)\}_{=1}^{mi}$
Output: $\eta_{j,k}$
1: Calculate the numbers of module pairs of each different rank, denoted as $\{m_{ij}, \ldots, m_{10}\}$;
2: $\eta_{j,k} = \dfrac{\max\ \{m_{j,k}, \ldots, m_{1,0}\}}{m_{j,k}}$.
3: **return** $\eta_{j,k}$;

is $\text{FPA}_{\text{correct}}$. For each module pair of $Rank_{5,1}$, we swap two modules. For example, if we swap the two modules in the module pair (M_0, M_1), then the new ranking is $M_1, M_0, M_2, M_3, M_4, M_5, M_6, M_7, M_8, M_9$ and the FPA value of the new ranking is FPA_{01}. In the same way, we swap the two modules in other module pairs of $Rank_{5,1}$. Then, we can obtain FPA_{02} and FPA_{03}. Finally, $\mu_{5,1} = \text{FPA}_{\text{correct}} - (\text{FPA}_{01} + \text{FPA}_{02} + \text{FPA}_{03})/3$. In the same way, we can calculate $\mu_{5,0}$ and $\mu_{1,0}$.

We use a simple method to calculate the value of $\mu_{j,k}$ (see Algorithm 2). We first calculate the numbers of module pairs of each different rank, denoted as $\{m_{ij}, \ldots, m_{10}\}$, where $m_{j,k}$ is the number of module pairs of rank $R_{j,k}$ (Line 1). Then, we define the value of $\eta_{j,k}$ as follows:

$$\eta_{\text{j,k}} = \frac{\max\ \{m_{j,k}, \ldots, m_{1,0}\}}{m_{j,k}}. \tag{8}$$

With such a parameter, we can place higher weight on the module pairs that occupy a very small proportion of the total loss function. As a result, the training is conducted equally over all module pairs.

Example 5: Consider the modules in Example 2, $m_{5,1} = 3$, $m_{5,0} = 6$, and $m_{1,0} = 18$. Thus, $\eta_{5,1} = 6$, $\eta_{5,0} = 3$, and $\eta_{1,0} = 1$.

B. Estimation of w

We use the genetic algorithm to estimate *w*, as it is commonly used by researchers in the field of software engineering

Algorithm 3: Estimation of ***w***.

Input: Module pairs dataset $S' = \{P_i = (\boldsymbol{p}_i, r_i)\}_{i=1}^{m}$
Number of solutions in a population, *PopSize*
Objective function, *L*(***w***)
Number of maximal generation, $t_{\max}$
Probability of crossover operator, p_c
Probability of mutation operator, p_m
Output: ***w***
1: Let P_0 = initial population with *PopSize* solutions;
2: Compute the objective function value of each solution in P_0;
3: Record the best solution found so far;
4: Set the current generation number $t = 1$;
5: **while** $t < t_{\max}$ **do**
6: $P_{\text{t}} = \text{select}(\text{P}_{t-1})$;
7: $P_t = \text{crossover}(P_t)$;
8: $P_t = \text{mutation}(P_t)$;
9: Compute the objective function value of each solution in P_t;
10: Record the best solution so far;
11: $t = t + 1$;
12: **end while**
13: **return** ***w***;

[57]–[63]. The genetic algorithm is a well-known search algorithm used to find true or approximate solutions to optimization problems, which convert the solution in a search space to a chromosome.

In this article, a solution is the value of *w*. The genetic algorithm starts with a set of randomly generated chromosomes, which is called the initial population. It then evolves the initial population by generating subsequent generations, where each generation is a population of chromosomes. The evolvement of the population contains four phases. First, in each generation, the fitness of every chromosome in the population is evaluated. Second, the selection phase selects the fittest chromosomes as the parent chromosomes to breed a new population. Third, the crossover phase changes the genes of the selected parent chromosomes according to a given probability p_c. Fourth, the mutation phase modifies the genes of new chromosomes with a given low probability p_m. The algorithm terminates if the population has converged or the maximum generation limit has been reached. Details on the genetic algorithm can be found in [64].

Algorithm 3 presents the pseudocode of estimating the value of *w* using the genetic algorithm. First, we randomly create an initial population P_0, which contains PopSize chromosomes (i.e., solutions) (Line 1). Then, we compute the objective loss function in formula (7) (i.e., the fitness function) and record the best solution (i.e., the solution with the minimum loss) (Lines 2 and 3). Next, we evolve the population through $t_{\max}$ iterations. In each iteration, we perform the selection, crossover, and mutation operations on the current population, and record the best solution found so far (Lines 5–12). Finally, the algorithm returns the *w* value which minimizes the loss function in formula (7) (i.e.,

TABLE II
DETAILS OF EXPERIMENTAL DATASETS

Project	Release	Module	%Defect	%1	%2-3	%4-5	%>5
Ant	1.3	125	16.0	8.8	7.2	0.0	0.0
	1.4	178	22.5	19.7	2.8	0.0	0.0
	1.5	293	10.9	9.9	1.0	0.0	0.0
	1.6	351	26.2	13.1	11.1	0.9	1.1
	1.7	745	22.3	12.5	6.2	2.6	1.1
Camel	1.0	339	3.8	3.5	0.3	0.0	0.0
	1.2	608	35.5	16.3	13.0	3.8	2.5
	1.4	872	16.6	8.1	6.0	1.3	1.3
	1.6	965	19.5	10.5	5.0	2.1	2.0
Ivy	1.1	111	56.8	18.9	21.6	8.1	8.1
	1.4	411	6.6	6.2	0.4	0.0	0.0
	2.0	352	11.4	8.0	3.4	0.0	0.0
Jedit	3.2	272	33.1	11.8	9.2	6.3	5.9
	4.0	306	24.5	11.8	8.2	1.3	3.3
	4.1	312	25.3	11.9	7.7	3.2	2.6
	4.2	367	13.1	7.4	3.8	0.5	1.4
	4.3	492	2.2	2.0	0.2	0.0	0.0
Log4j	1.0	135	25.2	15.6	8.1	0.7	0.7
	1.1	109	33.9	15.6	11.9	4.6	1.8
	1.2	205	92.2	8.8	68.3	11.2	3.9
Lucene	2.0	195	46.7	21.0	14.4	4.1	7.2
	2.2	247	58.3	24.3	22.7	5.7	5.7
	2.4	340	59.7	20.3	20.9	12.4	6.2
Poi	1.5	237	59.5	34.2	13.1	7.2	5.1
	2.0	314	11.8	11.1	0.6	0.0	0.0
	2.5	385	64.4	20.0	39.7	3.1	1.6
	3.0	442	63.6	45.5	12.9	2.9	2.3
Synapse	1.0	157	10.2	8.3	1.3	0.6	0.0
	1.1	222	27.0	18.0	7.2	1.4	0.5
	1.2	256	33.6	20.3	10.9	1.6	0.8
Velocity	1.4	196	75.0	53.1	19.9	1.5	0.5
	1.5	214	66.4	30.4	24.8	6.5	4.7
	1.6	229	34.1	14.8	13.5	2.2	3.5
Xalan	2.4	723	15.2	10.9	3.9	0.3	0.1
	2.5	803	48.2	37.0	9.3	1.4	0.5
	2.6	885	46.4	30.6	13.9	1.5	0.5
	2.7	909	98.8	72.6	24.6	1.1	0.4
Xerces	init	162	47.5	13.6	29.6	2.5	1.9
	1.2	440	16.1	7.0	8.6	0.5	0.0
	1.3	453	15.2	6.0	7.1	1.1	1.1
	1.4	588	74.3	30.8	26.4	4.6	12.6

the best solution among the population generated in the $t_{\max}$ generation) (Line 13). The parameters of the genetic algorithm are as follows: PopSize $= 100$, $t_{\max} = 100$, $p_c = 0.35$, and $p_m = 0.08$, since higher PopSize and $t_{\max}$ do not improve the performance of the CSRankSVM significantly.

V. EXPERIMENTAL SETUP

In this section, we detail the experimental setup. The experimental environment is a Windows 10 64-bit server with 8 GB RAM.

A. Datasets

Since we conduct the cross-version defect prediction in the experiment, we select the 11 public project datasets that include data for three or more versions from the PROMISE repository [66]. The choice of these datasets is also similar to Yang *et al.*'s article [5], which investigated the performance of some regression algorithms for RODP. Table II tabulates the details of the defect datasets, including the number of modules in the release (*Module*), the percentage of defect-prone modules in the release (*%Defect*), the percentage of modules with one defect in the release (%1), the percentage of modules with two defects and three defects in the release (%2–3), the percentage of modules with four and five defects in the release (%4–5), and the percentage of modules with more than five defects in the release ($\% > 5$).

TABLE III
FEATURES OF THE DATASETS

No.	Feature	Description
1	*wmc*	Weighted methods per class
2	*dit*	Depth of inheritance tree
3	*noc*	Number of children
4	*cbo*	Coupling between object classes
5	*rfc*	Response for a class
6	*lcom*	Lack of cohesion in methods
7	*ca*	Afferent couplings
8	*ce*	Efferent couplings
9	*npm*	Number of public methods
10	*lcom3*	Lack of cohesion in methods
11	*loc*	Lines of code
12	*dam*	Data access metric
13	*moa*	Measure of aggregation
14	*mfa*	Measure of functional abstraction
15	*cam*	Cohesion among methods of class
16	*ic*	Inheritance coupling
17	*cbm*	Coupling between methods
18	*amc*	Average method complexity
19	*max_cc*	Maximum McCabe's cyclomatic complexity
20	*avg_cc*	Average McCabe's cyclomatic complexity

It is worth noting that although the percentage between the defect-prone modules and the nondefective modules in Ivy 1.1, Log4j 1.2, Lucene, Poi 1.5, Poi 2.5, Poi 3.0, Velocity 1.4, Velocity 1.5, Xalan 2.5, Xalan 2.6, Xalan 2.7, Xerces init, and Xerces 1.4 is balanced, the number of defects in each module in these datasets is highly imbalanced. That is, the modules with many defects occupy only a small part of this project, whereas the defect-free modules occupy a great part of this project, followed by the modules with one defect. In addition, each module in the 11 projects has the same 20 independent code attributes, including the lines of code, weighted methods per class, and depth of inheritance tree. A more detailed description of the 20 independent code attributes is listed in Table III.

B. Performance Measures

Menzies *et al.* [67], Kamei *et al.* [68], and D'Ambros *et al.* [69] pointed out that effort should be taken into consideration when testing the modules suggested by the defect prediction models. Traditional performance measures used for SDP (precision, recall, F-measure, and AUC) are not well suited to the evaluation of the RODP models, as they give the same importance to all defective software modules. Therefore, some researchers have proposed some performance measures for evaluating the performance of the SDP models in cost-sensitive scenarios, such as cost effectiveness (CE) [70], *Norm*(P_{opt}) [71], CLC [72], and the FPA [73]. The former two performance measures are SLOC based (i.e., they evaluate how many defects can be found when we inspect a certain line of code), whereas the latter two performance measures are module based (i.e., they evaluate how many defects can be found when we inspect a certain number of

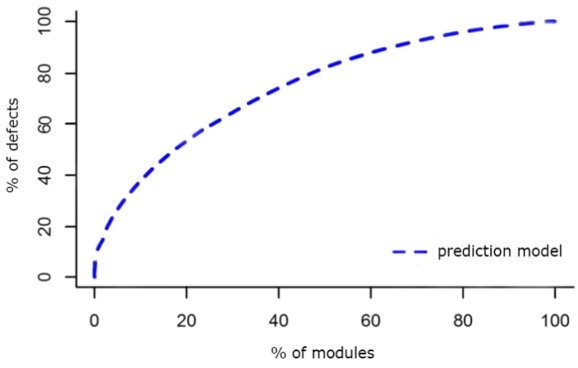

Fig. 2. Module-based CLC.

modules). Ostrand *et al.* [46] pointed out that software testers are more concerned with modules that contain more defects and that using SLOC to evaluate the CE of the SDP models is not valid. Therefore, we use the module-based performance measures. We explain the two module-based performance measures as follows.

1) In the module-based CLC (see Fig. 2), the *x*-axis is the cumulative percentage of modules to inspect and the *y*-axis is the cumulative percentage of defects. The CLC consists of a curve of the prediction model, where all modules are ordered by decreasing predicted number of defects. The CLC is defined as the area under the curve.
2) The FPA is the average of the proportions of actual defects in the top modules to the total defects. Assume that *n* modules in a project are ranked by the nondecreasing order of the predicted number of defects, as $M_1, M_2, M_3, \ldots, M_n$, and $Y = y_1 + y_2 + \cdots + y_n$ is the total number of defects in these modules. Therefore, M_n is predicted to contain the most number of defects. The proportion of the actual defects in the top *m* predicted modules to the total defects is calculated as follows:

$$\frac{1}{Y}\sum_{i=n-m+1}^{n} y_i. \tag{9}$$

Then, the FPA is defined as follows:

$$\frac{1}{n}\sum_{m=1}^{n}\frac{1}{Y}\sum_{i=n-m+1}^{n} y_i. \tag{10}$$

Yang *et al.* [5] proved that the CLC and FPA are linearly related. In the experiment, we use the FPA to measure the performance, as the FPA is the up-to-date performance measure proposed to evaluate the performance of the RODP models and its formula is easier to understand [5].

C. Research Questions

In this experiment, we investigate the following five research questions.

RQ1. Does the CSRankSVM outperform state-of-the-art RODP methods?

To answer this question, we compare the CSRankSVM with three regression-based RODP methods (DTR [74], BRR [75], and LR [76]), as these algorithms achieve better performance in most cases in predicting the number of defects [54], [55]. We also compare the CSRankSVM with two learning to rank methods: learning-to-rank (LTR) [5] and ranking SVM [13]. These methods are described in Section IV-D.

RQ2. Are the proposed cost-sensitive strategies in the CSRankSVM more effective than other data imbalance learning methods?

As mentioned in Section II-A, three types of approaches can deal with data imbalance (i.e., data sampling, ensemble learning, and cost-sensitive learning). Data sampling techniques include oversampling and undersampling. Oversampling adds synthetic defective modules to achieve the class-balanced state, whereas undersampling removes nondefective modules. We apply both RUS and the SMOTE to the defect datasets to obtain the balanced datasets and compare the CSRankSVM with the baseline methods in RQ1 trained on the balanced datasets. In addition, we compare the CSRankSVM with RankBoost (an ensemble learning method) [14], IRSVM (a cost-sensitive learning ranking SVM method for information retrieval) [77], and the CSRankSVM (a cost-sensitive learning ranking SVM method with only the penalty parameter $\eta_{j,k}$). These methods are described in Section IV-D.

RQ3. How effective are the two cost parameters (i.e., $\mu_{j,k}$ and $\eta_{j,k}$) of the CSRankSVM?

To demonstrate the capability of the two cost parameters of the CSRankSVM, we investigate third subquestions: (RQ3a) Are more defective modules in the top 20% ranking list of all modules after adding the two cost parameters? (RQ3b) Are more defects found by inspecting the top 20% ranking list of all modules after adding the two cost parameters? (RQ3c) Are the modules with most defects ranked higher after adding the two cost parameters?

RQ4. Which software features are more effective for the CSRankSVM?

To answer this question, we use information gain (IG) as the feature selection method to investigate the effectiveness of different features. IG is an entropy-based method, which measures the reduction of uncertainty about the target variable value after observing the feature. We use IG for two reasons. First, some empirical studies [86] have demonstrated its effectiveness for SDP. Second, the work on feature selection for RODP has been limited. Only Yang *et al.* [5] applied IG to investigate the effectiveness of different features for RODP. Following the setup in article [5], we adopt the iterative subset by selecting the 2, 3, 5, 8, and 13 top-ranked features.

RQ5. What is the execution time for the CSRankSVM?

The CSRankSVM uses the genetic algorithm to optimize the modified loss function, which is time consuming. Therefore, we investigate the time efficiency of the CSRankSVM in this research question. We compare the training time and test time of the CSRankSVM with those of DTR, LR, BRR, ranking SVM, LTR, RankBoost, IRSVM, and CSRankSVM-.

D. Methods in Comparison

To answer RQ1 and RQ2, we compare the CSRankSVM with three regression-based RODP methods (DTR, BRR, and LR),

two learning to rank methods (Ranking SVM and LTR), and two data-sampling methods (RUS and SMOTE), an ensemble learning method (RankBoost), and a cost-sensitive learning method (IRSVM). These methods are briefly described as follows.

1) DTR: The DTR method builds a regression model in the form of a decision tree structure by learning from the training dataset. A decision tree is built top–down from a root node and brakes down the training dataset into increasingly smaller subsets with a splitting criterion. The final result is a tree with leaf nodes and decision nodes.
2) LR: The LR method is a statistical method to build a linear relationship between the number of defects and the software features. An LR model can be described according to the following equation:

$$y = \langle \boldsymbol{b}, \boldsymbol{x} \rangle + b_0 \tag{11}$$

where $\boldsymbol{b} = (b_1, b_2, \ldots, b_d)$ denotes a vector of regression coefficients and b_0 is the error term.

3) BRR: The BRR method is a probabilistic method for building a regression model using Bayesian inference. It combines prior information about parameters (the coefficient of software features) with the observed training data to obtain the posterior distribution of the parameters.
4) LTR: The LTR method trains a simple linear model

$$f(x) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle . \tag{12}$$

It uses the composite differential evolution algorithm to directly optimize the FPA value to obtain $\boldsymbol{w}$. It then uses the trained model to predict the relative number of defects in new modules and ranks these modules based on the predicted values.

5) RUS: The RUS method randomly deletes nondefective modules to balance the ratio of defective modules to nondefective modules. In the experiment, we set the desired ratio to 100% to obtain a balanced dataset. We first apply RUS to the defect dataset to obtain a balanced dataset and then train DTR, LR, BRR, ranking SVM, and LTR based on the balanced dataset.
6) SMOTE: The SMOTE method first randomly chooses a defective module, then finds its k nearest neighbors, and finally generates a synthetic defective module based on the module and one of its k neighbors. We use the same approach in [65] to decide the number of defects in the generated module. That is, the number of defects in the generated module is the weighted average of the numbers of defects in the two seed modules. The weights are calculated according to the distance between the synthetic module and the two seed modules. The smaller the distance is, the larger the weight is. In the experiment, we set k to 5 and the desired ratio between the defective modules and the nondefective modules to 100% to obtain a balanced dataset. We first apply SMOTE to the defect dataset to obtain a balanced dataset and then train DTR, LR, BRR, ranking SVM, and LTR based on the balanced dataset.
7) RankBoost: The RankBoost method is a boosting method for ranking. Similar to all boosting algorithms, it trains a weak ranking learner at each iteration and then combines these weak ranking learners as the final ranking learner. The weak ranking learner h is derived from a ranking feature f_i by comparing the score of f_i on a given software module x to a threshold θ. It is defined as follows:

$$h(x) = \begin{cases} 1, & \text{if } f_i(\boldsymbol{x}) > \theta \\ 0, & \text{if } f_i(\boldsymbol{x}) \ll \theta \\ q_{\text{def}}, & \text{if } f_i(\boldsymbol{x}) = \perp \end{cases} \tag{13}$$

where if $f_i(\boldsymbol{x}) = \perp$, then $h(x) = q_{\text{def}}$ means that if the software module is unranked by f_i, the weak ranking learner assigns the default score q_{def}. Then, the "best" feature, threshold and default score, is solved by minimizing the loss function of the algorithms.

8) IRSVM: The IRSVM method is a CSRankSVM method for information retrieval. Specifically, it sets different losses for the misclassification of instance pairs between different rank pairs. That is, the IRSVM takes the cost issue into consideration, but not the data imbalance problem.
9) CSRankSVM-: The CSRankSVM- is a cost-sensitive learning ranking SVM method with only the penalty parameter $\eta_{j,k}$. That is, the CSRankSVM- takes the data imbalance problem into consideration, but not the cost issue.

Therefore, we can investigate whether the performance of the CSRankSVM will decrease if we remove any one of the penalty parameters by comparing the CSRankSVM with the IRSVM and CSRankSVM-.

E. Experimental Design Summary

Tantithamthavorn *et al.* [78] performed an empirical study on validation techniques for SDP and recommended out-of-sample bootstrap validation, so all of the experiments are performed using out-of-sample bootstrap, except that we also use the cross-version setting for RQ1. A bootstrap sample of size N is randomly drawn with replacement from a dataset with N modules. The RODP models are then trained on the bootstrap sample and tested on the modules in the original dataset that are not contained in the bootstrap sample. The procedure is repeated 20 times for the same set of data. As a result, we obtain 20 results for each method over each set of training and testing data. We list the median value of the 20 results in the tables, on which bold font highlights the best performance in the row.

Then, we perform the Wilcoxon signed-rank test [79] to analyze the significance of the differences between median results achieved by the two methods over all datasets. We also use the Benjamini—Hochberg (BH) procedure to adjust p-values since we perform multiple comparisons [80]. If the BH corrected p-value is less than 0.05, it means that there is a statistically significant difference between the two methods. In addition, we calculate the effect size (i.e., Cliff's δ) to measure the differences between median results achieved by the two methods over all datasets. By convention, the magnitude of the difference is considered negligible ($0 <$ Cliff's $\delta < 0.147$), small ($0.147 <$ Cliff's $\delta < 0.33$), medium ($0.33 <$ Cliff's $\delta < 0.474$), or large (Cliff's $\delta > 0.474$) [81].

TABLE IV
FPA VALUES OF THE DIFFERENT METHODS (THE BEST FPA VALUES FOR EACH DATASET ARE IN BOLD)

Dataset	CSRankSVM	DTR	LR	BRR	Ranking SVM	LTR
Ant 1.3	**0.698**	0.604	0.631	0.631	0.590	0.684
Ant 1.4	**0.657**	0.544	0.631	0.626	0.586	0.591
Ant 1.5	**0.816**	0.597	0.768	0.809	0.681	0.772
Ant 1.6	0.787	0.686	0.785	**0.809**	0.718	0.793
Ant 1.7	**0.821**	0.685	0.808	0.818	0.654	0.812
Camel 1.0	**0.715**	0.579	0.572	0.654	0.498	0.590
Camel 1.2	**0.668**	0.596	0.658	0.660	0.594	0.644
Camel 1.4	0.672	0.592	0.674	0.691	0.610	**0.710**
Camel 1.6	**0.749**	0.612	0.744	0.739	0.571	0.722
Ivy 1.1	0.794	0.746	0.764	0.791	0.472	**0.804**
Ivy 1.4	**0.737**	0.672	0.575	0.552	0.437	0.545
Ivy 2.0	0.791	0.626	0.815	0.802	0.745	**0.816**
Jedit 3.2	**0.843**	0.730	0.837	0.840	0.603	0.840
Jedit 4.0	0.794	0.751	**0.830**	0.809	0.791	0.811
Jedit 4.1	0.789	0.714	0.803	**0.807**	0.699	0.799
Jedit 4.2	0.861	0.708	**0.866**	0.857	0.784	0.843
Jedit 4.3	0.828	0.420	0.696	**0.874**	0.553	0.548
Log4j 1.0	**0.800**	0.706	0.684	0.781	0.684	0.742
Log4j 1.1	0.792	0.731	0.775	**0.814**	0.702	0.791
Log4j 1.2	0.575	0.552	0.568	**0.576**	0.548	0.573
Lucene 2.0	0.736	0.697	0.739	0.747	0.572	**0.751**
Lucene 2.2	**0.680**	0.635	0.653	0.674	0.581	0.668
Lucene 2.4	0.713	0.658	0.716	**0.717**	0.623	0.692
Poi 1.5	**0.657**	0.609	0.641	0.650	0.581	0.652
Poi 2.0	**0.732**	0.611	0.569	0.641	0.623	0.677
Poi 2.5	0.638	**0.662**	0.648	0.653	0.642	0.640
Poi 3.0	0.667	0.652	0.687	**0.696**	0.628	0.693
Synapse 1.0	0.701	0.664	0.753	**0.809**	0.613	0.731
Synapse 1.1	0.707	0.682	**0.737**	0.730	0.680	0.712
Synapse 1.2	0.676	0.614	0.665	**0.710**	0.650	0.673
Velocity 1.4	0.615	0.601	0.621	0.612	0.585	**0.773**
Velocity 1.5	**0.690**	0.625	0.651	0.669	0.639	0.654
Velocity 1.6	**0.754**	0.673	0.689	0.680	0.612	0.691
Xalan 2.4	**0.756**	0.591	0.730	0.752	0.719	0.560
Xalan 2.5	0.609	0.575	0.632	**0.642**	0.533	0.592
Xalan 2.6	**0.694**	0.638	0.675	0.686	0.632	0.669
Xalan 2.7	0.579	0.537	0.557	0.576	0.578	**0.671**
Xerces init	0.690	0.682	0.621	0.611	0.602	**0.733**
Xerces 1.2	**0.685**	0.583	0.651	0.602	0.565	0.623
Xerces 1.3	0.758	0.725	0.777	**0.784**	0.744	0.772
Xerces 1.4	0.735	0.720	0.734	0.744	0.719	**0.751**
Avg.	0.723	0.641	0.698	0.715	0.625	0.703
W/D/L		40/0/1	27/0/14	23/0/18	40/0/1	25/0/16
p-value		0.000	0.008	0.816	0.000	0.125
Cliff's δ		0.587	0.203	0.066	0.636	0.140

VI. EXPERIMENTAL RESULTS

A. RQ1

To answer RQ1, we compare the CSRankSVM with DTR, LR, BRR, Ranking SVM, and LTR. Table IV presents the detailed FPA values of the six methods. Considering the average FPA value, the CSRankSVM obtains the best performance among the six methods. In addition, the CSRankSVM achieves the highest FPA value on 18 datasets. The win/draw/loss (W/D/L) row presents the results for the number of datasets, on which the CSRankSVM performs better than, the same as, or worse than the other methods. The W/D/L values in Table IV show that compared with DTR, LR, BRR, ranking SVM, and LTR, the CSRankSVM wins on 40, 27, 23, 40, and 25 datasets, respectively. In addition, the CSRankSVM improves the average FPA value of DTR by 12.78%, of LR by 3.58%, of BRR by 1.12%, of ranking SVM by 15.68%, and of LTR by 2.84%, respectively. In addition, the corrected p-values show that there is a statistically significant difference between the CSRankSVM and DTR, LR, and ranking SVM (p-value < 0.05). Cliff's δ values indicate that the performance of the CSRankSVM has a larger effect than those of DTR and ranking SVM (Cliff's $\delta > 0.474$). Cliff's δ values for LR is 0.203, which can also be considered as small effects (i.e., greater than 0.147 but less than 0.33).

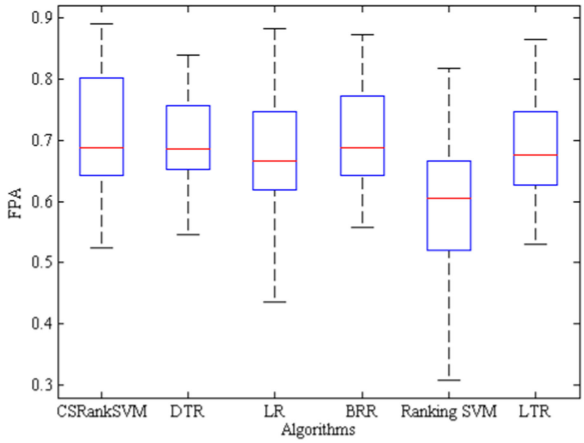

Fig. 3. Boxplot of the FPA values with the six methods using cross-version setting.

In addition, testers may use a cross-version validation setting (i.e., using the prior version as the training dataset and the current version as the testing dataset). Therefore, we also investigate the effectiveness of the CSRankSVM when using the cross-version setting. Due to the space limit, we only plot the boxplot of the entire distribution of the FPA values across all cross-version pairs in Fig. 3. As shown in Fig. 3, the median value by the CSRankSVM is higher than all of the compared methods. In addition, the maximum value by the CSRankSVM is much higher than the maximum values achieved by the other algorithms.

From Table IV and Fig. 3, we also find that BRR and LTR perform well. Indeed, the CSRankSVM is not significantly better than BRR and LTR in most situations in terms of the average FPA, but the CSRankSVM wins BRR and LTR on more than half of the datasets. Therefore, we still recommend CSRankSVM as an effective method for RODP. In addition, ranking SVM performs worst, because it aims to minimize the number of incorrect rankings and does not account for the cost issue and the data imbalance problem. However, after modifying the loss function of ranking SVM, the CSRankSVM gains great

TABLE V
FPA VALUES OF THE DIFFERENT DATA IMBALANCE LEARNING METHODS (THE BEST FPA VALUES FOR EACH DATASET ARE IN BOLD)

Datasets	CSRankSVM	RUS+ DTR	RUS+ LR	RUS+ BRR	RUS+ Ranking SVM	RUS+ LTR	SMOTE+ DTR	SMOTE+ LR	SMOTE+ BRR	SMOTE+ Ranking SVM	SMOTE+ LTR	RankBoost	IRSVM	CSRankSVM-
Ant 1.3	0.698	0.605	0.554	0.641	0.510	0.701	0.649	0.670	0.698	0.572	**0.713**	0.662	0.623	0.588
Ant 1.4	0.657	0.568	0.604	0.589	0.583	0.579	0.602	0.648	**0.679**	0.628	0.616	0.636	0.663	0.622
Ant 1.5	**0.816**	0.670	0.671	0.782	0.629	0.736	0.566	0.761	0.781	0.653	0.775	0.754	0.774	0.760
Ant 1.6	0.787	0.708	0.773	**0.810**	0.682	0.802	0.691	0.790	0.806	0.674	0.798	0.804	0.747	0.739
Ant 1.7	**0.821**	0.699	0.792	0.812	0.745	0.810	0.701	0.802	0.809	0.744	0.806	0.797	0.773	0.772
Camel 1.0	**0.715**	0.567	0.505	0.700	0.559	0.638	0.578	0.573	0.598	0.603	0.629	0.652	0.594	0.519
Camel 1.2	**0.668**	0.585	0.660	0.661	0.659	0.642	0.580	0.666	0.660	0.605	0.644	0.666	0.590	0.572
Camel 1.4	0.672	0.627	0.680	0.713	0.595	**0.734**	0.618	0.695	0.702	0.601	0.732	0.715	0.682	0.655
Camel 1.6	**0.749**	0.622	0.738	0.738	0.600	0.730	0.636	0.742	0.738	0.625	0.724	0.724	0.684	0.618
Ivy 1.4	**0.737**	0.568	0.552	0.726	0.600	0.651	0.639	0.545	0.555	0.512	0.570	0.630	0.413	0.444
Ivy 2.0	0.791	0.703	0.742	**0.828**	0.727	0.814	0.664	0.786	0.803	0.778	0.804	0.801	0.729	0.724
Jedit 3.2	**0.843**	0.729	0.826	0.835	0.706	0.827	0.725	0.839	0.839	0.536	0.834	0.835	0.800	0.789
Jedit 4.0	0.794	0.755	0.809	0.827	0.724	0.830	0.770	**0.837**	0.832	0.711	0.814	0.835	0.770	0.735
Jedit 4.1	0.789	0.714	0.777	**0.811**	0.688	0.802	0.712	0.802	0.809	0.680	0.806	0.779	0.761	0.764
Jedit 4.2	0.861	0.766	0.836	**0.862**	0.791	0.840	0.753	0.859	0.860	0.788	0.838	0.837	0.810	0.760
Jedit 4.3	**0.828**	0.556	0.450	0.649	0.631	0.564	0.423	0.637	0.650	0.680	0.728	0.735	0.596	0.452
Log4j 1.0	**0.800**	0.700	0.658	0.793	0.605	0.755	0.705	0.708	0.751	0.701	0.755	0.765	0.686	0.694
Log4j 1.1	0.792	0.749	0.771	0.817	0.780	0.804	0.733	0.792	0.809	0.722	0.788	**0.827**	0.753	0.757
Poi 2.0	0.638	**0.661**	0.648	0.653	0.617	0.634	0.660	0.648	0.653	0.631	0.643	0.653	0.617	0.621
Synapse 1.0	0.701	0.710	0.601	**0.807**	0.594	0.780	0.597	0.745	0.777	0.709	0.769	0.699	0.707	0.615
Synapse 1.1	0.707	0.710	0.600	**0.807**	0.594	0.780	0.596	0.745	0.776	0.708	0.769	0.730	0.696	0.687
Synapse 1.2	0.676	0.629	0.661	0.702	0.625	0.692	0.623	0.674	**0.705**	0.574	0.682	0.671	0.659	0.658
Velocity 1.6	**0.754**	0.631	0.693	0.677	0.705	0.686	0.668	0.711	0.676	0.674	0.687	0.665	0.673	0.695
Xalan 2.4	0.756	0.622	0.705	0.747	0.731	**0.775**	0.605	0.742	0.745	0.666	0.764	0.552	0.732	0.661
Xerces 1.2	**0.685**	0.632	0.632	0.534	0.579	0.609	0.621	0.650	0.635	0.611	0.628	0.603	0.639	0.609
Xerces 1.3	0.758	0.730	0.763	0.778	0.660	0.763	0.732	**0.789**	0.787	0.679	0.772	0.784	0.729	0.730
Avg	0.750	0.662	0.681	0.742	0.651	0.730	0.648	0.725	0.736	0.656	0.734	0.723	0.688	0.663
W/D/L		23/0/3	22/0/4	14/0/12	26/0/0	14/0/12	25/0/1	17/1/8	13/1/12	24/0/2	14/0/12	18/0/8	23/0/3	26/0/0
p-value		0.000	0.000	1.000	0.000	0.269	0.000	0.191	0.945	0.000	0.277	0.075	0.000	0.000
Cliff's δ		0.637	0.416	-0.001	0.691	0.109	0.715	0.180	0.087	0.667	0.088	0.195	0.426	0.543

performance improvement, which indicates that cost-sensitive learning can be a good solution to learn from imbalanced data for RODP.

According to the experimental results in Table IV and Fig. 3, we conclude that the CSRankSVM outperforms the existing methods in terms of FPA.

B. RQ2

To answer RQ2, we compare the CSRankSVM with two data-sampling methods (RUS and SMOTE), an ensemble learning method (RankBoost), and the two cost-sensitive learning methods (IRSVM and CSRankSVM-) using the bootstrap setting. As the percentage of defect-prone modules and nondefective modules in Ivy 1.1, Log4j 1.2, Lucene, Poi 1.5, Poi 2.5, Poi 3.0, Velocity 1.4, Velocity 1.5, Xalan 2.5, Xalan 2.6, Xalan 2.7, Xerces init, and Xerces 1.4 is balanced, we do not apply RUS and SMOTE to these datasets. We apply RUS and SMOTE to other datasets to obtain the balanced datasets and train DTR, LR, BRR, ranking SVM, and LTR on the balanced datasets. We call the methods trained on the balanced datasets RUS+DTR, RUS+LR, RUS+BRR, RUS+Ranking SVM, RUS+LTR, SMOTE+DTR, SMOTE+LR, SMOTE+BRR, SMOTE+Ranking SVM, and SMOTE+LTR. Similar to RQ1, Table V presents the detailed FPA values of these methods.

Comparing the FPA results in Table IV and those in Table V, we observe that for DTR, BRR, ranking SVM and LTR, there are 2.80%, 0.67%, 1.24%, and 2.82% performance improvements in terms of the average FPA values, after applying RUS to the datasets. However, the FPA values for LR decrease after applying RUS to the datasets. The CSRankSVM achieves 1.08%–15.21% higher average FPA values than the compared methods after applying RUS. After applying SMOTE to the datasets, the average FPA values of DTR, LR, ranking SVM, and LTR increase 0.62%, 0.28%, 2.02%, and 3.38% respectively. But for BRR, there is a little performance degradation. CSRankSVM achieves 1.90%–15.74% higher FPA values than the compared methods after applying SMOTE. The W/D/L values show that compared with RUS+DTR, RUS+LR, RUS+Ranking SVM, and SMOTE+LTR, SMOTE+DTR, SMOTE+LR, SMOTE+Ranking SVM, and SMOTE+LTR, the CSRankSVM wins on 23, 22, 14, 26, 14, 25, 17, 13, 24, and 14 datasets, respectively.

In addition, the BH corrected p-values show that there is a statistical significant difference between the CSRankSVM and RUS+DTR, RUS+LR, RUS+Ranking SVM, SMOTE+DTR, and SMOTE+Ranking SVM (p-value < 0.05). Cliff's δ values indicate that the performance of the CSRankSVM has a large effect than those of RUS+DTR, RUS+Ranking SVM, SMOTE+DTR, and SMOTE+Ranking SVM (Cliff's $\delta > 0.474$).

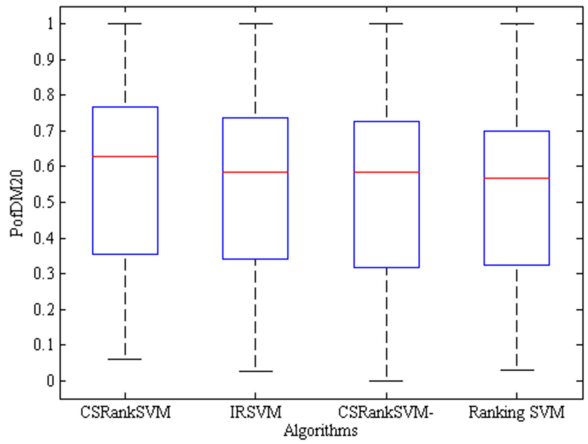

Fig. 4. Boxplot of the PofDM20 values with the four methods.

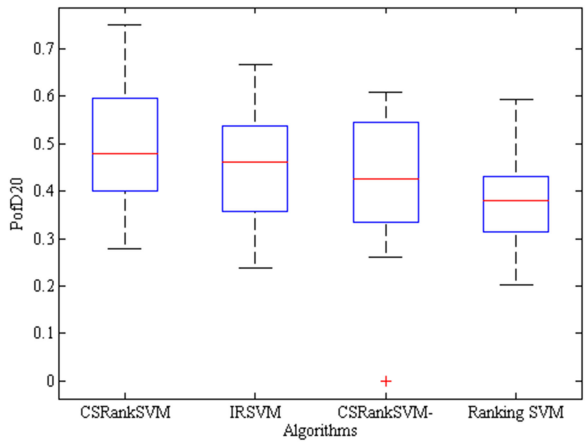

Fig. 5. Boxplot of the PofD20 values with the four methods.

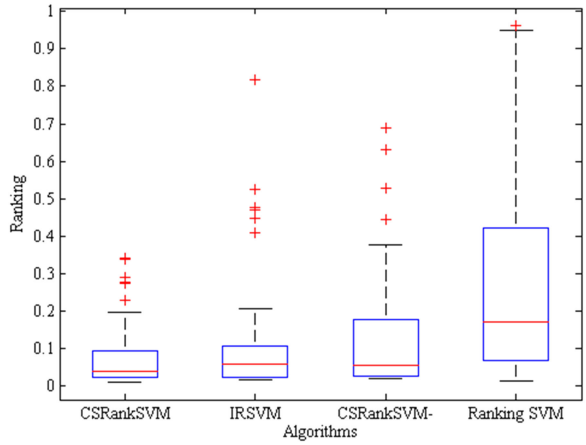

Fig. 6. Boxplot of the ranking values with the four methods.

Compared with RankBoost, IRSVM, and CSRankSVM-, the CSRankSVM achieves the highest average FPA value and wins on 18, 23, and 26, datasets, respectively. The CSRankSVM improves the average FPA value of RankBoost by 3.73%, of IRSVM by 9.01%, and the CSRankSVM- by 13.12%. There are statistically significant differences between the CSRankSVM and the two cost-sensitive learning methods (p-value < 0.05). Cliff's δ values show that the CSRankSVM has a large effect compared with IRSVM and CSRankSVM-, respectively. In addition, comparison of the CSRankSVM, IRSVM, and CSRankSVM- shows that the performance of the CSRankSVM decreases if we remove any one of the penalty parameters. This indicates that both penalty parameters help improve the prediction performance.

According to the experimental results in Table V, we conclude that the CSRankSVM outperforms RUS and SMOTE (when employing DTR, LR, BRR, ranking SVM, and LTR as the RODP methods), an ensemble learning method (RankBoost), and the two cost-sensitive learning methods (IRSVM and CSRankSVM-). In other words, the cost-sensitive strategy of the CSRankSVM is more effective than the compared data imbalance learning methods.

C. RQ3

To demonstrate the capability of the two cost parameters (i.e., $\mu_{j,k}$ and $\eta_{j,k}$) of the CSRankSVM, we first investigate whether more defective modules are in the top 20% ranking list and how many defects can be discovered by inspecting the top 20% modules after adding the two cost parameters, we use the PofDM20 and PofD20 as the performance measure. The PofDM20 is the percentage of defective modules that can be discovered by inspecting the top 20% ranking list of all modules. The PofD20 is the percentage of defects that can be discovered by inspecting the top 20% ranking list of all modules.

Figs. 4 and 5 present the boxplots of the entire distribution of the PofDM20 and PofD20 values across all datasets, respectively. As shown in Figs. 4 and 5, the median value by the CSRankSVM is higher than all of the compared methods in terms of PofDM20 and PofD20. In addition, the maximum value by the CSRankSVM is much higher than the maximum values achieved by the other algorithms in terms of PofD20. Therefore, we can conclude that the two cost parameters of the CSRankSVM can contribute to make more defective modules rank in the top 20% ranking list of all modules and finding more bugs.

Then, in order to investigate whether the module with most defects is ranked higher after adding the two cost parameters, we use Ranking as the performance measure, which is the ranking of the module with most defects to the whole ranking list. For example, if the whole ranking list contains 10 modules and the module with most defects is ranked 2nd in the whole ranking list, then Ranking is 0.2 (=2/10).

Fig. 6 presents the boxplot of the entire distribution of the ranking values across all datasets, respectively. As shown in Fig. 6, the median value by the CSRankSVM is lower than all of the compared methods. In addition, the maximum value by the CSRankSVM is much lower than the maximum values achieved by the other algorithms. Therefore, we can conclude that the two cost parameters of the CSRankSVM can contribute to make the module with most defects rank higher.

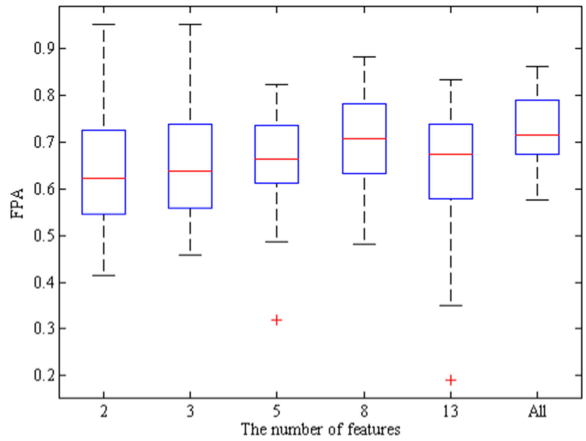

Fig. 7. Boxplot of the FPA values with CSRankSVM trained on the different number of features.

D. RQ4

To answer RQ4, we use IG as the feature selection method to investigate the effectiveness of different features. IG is an entropy-based method. IG measures the reduction of uncertainty about the target variable value after observing the feature. We use IG for two reasons. First, some empirical studies [86] have demonstrated its effectiveness for SDP. Second, the work on feature selection for RODP has been limited. Only Yang *et al.* [5] applied IG to investigate the effectiveness of different features for RODP. Following the setup in paper [5], we adopt the iterative subset by selecting the 2, 3, 5, 8, and 13 top-ranked features.

Fig. 7 presents the boxplot of the entire distribution of the FPA values based on different numbers of features. As shown in Fig. 7, the median and maximum values of the CSRankSVM trained on all features are higher than those of the CSRankSVM trained on fewer features. That is, models based on fewer features do not work better than models based on all features, which indicates that IG does not improve the performance of the CSRankSVM. This observation is not consistent with previous studies [86]–[88]. One reason may be that IG is applicable to classification tasks, but not to ranking tasks. Therefore, designing a more effective feature selection method for RODP is one of our future research interests.

E. RQ5

Table VI lists the average training time and testing time over the 11 project datasets in Table II. To conserve space, we do not list the time for each dataset. As given in Table VI, we observe that the CSRankSVM requires 764.32 s to train a model and 0.21 s to test it. In addition, the training times of LTR, RankBoost, IRSVM, and CSRankSVM- are 98.91 s, 157.32 s, 725.46 s, and 732.87 s, respectively, as these methods also require multiple iterations to obtain the optimized training models. Although, the training time of the CSRankSVM is a little long, the testing time is less than 1 s. Therefore, we argue that it is still acceptable.

TABLE VI
AVERAGE TRAINING TIME AND TESTING TIME OF DIFFERENT METHODS

Project	Training Time	Test Time
CSRankSVM	764.32 s	0.21 s
DTR	0.03 s	0.0001 s
LR	0.26 s	0.0001 s
BRR	0.045 s	0.0001 s
Ranking SVM	0.31 s	0.039 s
LTR	98.91 s	0.01 s
RankBoost	157.32 s	0.31 s
IRSVM	725.46 s	0.45 s
CSRankSVM-	732.87 s	0.46 s

VII. THREATS TO VALIDITY

In this section, we discuss some potential threats to validity that may affect the results of this article.

In our experiments, we choose 11 project datasets from the PROMISE repository. Although they have been widely used [82]–[84], we still cannot claim that our method would perform best for other defect datasets, especially proprietary datasets, which we do not consider in this article.

We set the parameters for the baseline methods according to the existing works or default parameters. The best parameters for different datasets may be different, which may result in different results. We set the desired ratio between the defective modules and the nondefective modules to 100% when applying RUS and SMOTE to the training datasets, as this is common practice. The optimal ratios for the different datasets and different baseline methods are different, so it is difficult to choose the fixed optimal desired ratio. The proposed approach assumes that software quality teams allocate testing resources based on the number of defects, ignoring the severity of defects and the testing time. This may not always be the case. Thus, our method can be improved by considering the severity of the defects.

In our experiments, we use the FPA as the evaluation measure, as it is a module-based performance measure. Furthermore, Ostrand *et al.* [46] acknowledged that software testers are more concerned with the modules that contain more defects. Additionally, the FPA is the up-to-date performance measure, which is linearly related to the CLC and is easier to understand [5].

VIII. CONCLUSION

In this article, we proposed CSRankSVM, a cost-sensitive learning method that learns from imbalanced defect data for RODP. The RODP models rank software modules based on the predicted number of defects and consequently help in the efficient allocation of testing resources. Apparently, the actual cost of incorrectly ranking a module with many defects was much higher than the cost of incorrectly ranking a module with one defect, thus it was necessary to incorporate the incorrect ranking costs into the RODP models. Additionally, defect datasets were known to be highly class imbalanced. However, existing RODP methods did not take these two issues into consideration. Therefore, the CSRankSVM incorporates two cost parameters

(i.e., $\mu_{j,k}$ and $\eta_{j,k}$) into the loss function used in ranking SVM to address the cost and data imbalance issues. We then used a genetic algorithm to optimize the loss function. In the experiment, we empirically evaluated the performance of the CSRankSVM on 11 open-source project datasets with 41 releases. The results showed that the CSRankSVM outperforms the baseline methods, including the five existing RODP methods and four data imbalance learning methods. The experimental results showed that the two cost parameters of the CSRankSVM could solve the cost issue and the data imbalance problem, and achieved better performance. We also observed that the two cost parameters of the CSRankSVM made more defective modules rank in the top 20% ranking list of all modules, and made the module with most defects rank higher. Therefore, the CSRankSVM was recommended as an effective method for RODP. Apart from the CSRankSVM, we have discovered that (Bayesian ridge regression (BRR) also achieves good performance. Therefore, designing an effective data imbalance learning method based on BRR was also a potential direction for future improvement for RODP.

REFERENCES

[1] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 61–72.

[2] K. Gao, T. M. Khoshgoftaar, and H. Wang, "Choosing software metrics for defect prediction: An investigation on feature selection techniques," *Softw. Pract. Experience*, vol. 41, no. 5, pp. 579–606, 2011.

[3] X. Yang and W. Wen, "Ridge and lasso regression models for cross-version defect prediction," *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 885–896, Sep. 2018.

[4] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect proneness prediction framework," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 356–370, May/Jun. 2011.

[5] X. Yang, K. Tang, and X. Yao, "A learning-to-rank approach to software defect prediction," *IEEE Trans. Rel.*, vol. 64, no. 1, pp. 234–246, Mar. 2015.

[6] F. Wu *et al.*, "Cross-project and within-project semisupervised software defect prediction: A unified approach," *IEEE Trans. Rel.*, vol. 67, no. 2, pp. 581–597, Jun. 2018.

[7] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Appl. Soft Comput.*, vol. 27, pp. 504–518, 2015.

[8] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 40, no. 6, pp. 603–616, Jun. 2014.

[9] W. Liu, S. Liu, Q. Gu, J. Chen, X. Chen, and D. Chen, "Empirical studies of a two-stage data preprocessing approach for software fault prediction," *IEEE Trans. Rel.*, vol. 65, no. 1, pp. 38–53, Mar. 2016.

[10] H. Lu, E. Kocaguneli, and B. Cukic, "Defect prediction between software versions with active learning and dimensionality reduction," in *Proc. 25th Int. Symp. Softw. Rel. Eng.*, 2014, pp. 312–322.

[11] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 675–689, Sep./Oct. 1999.

[12] T. T. Nguyen *et al.*, "Similarity-based and rank-based defect prediction," in *Proc. Int. Conf. Adv. Technol. Commun.*, 2015, pp. 321–325.

[13] R. Herbrich, T. Graepel, and K. Obermayer, "Large margin rank boundaries for ordinal regression," in *Adv. Large Margin Classifiers*, 2000, pp. 115–132.

[14] Y. Freund *et al.*, "An efficient boosting algorithm for combining preferences," in *Proc. 15th Int. Conf. Mach. Learn.*, 1998, pp. 170–178.

[15] M. M. T. Thwin and T. S. Quah, "Application of neural networks for software quality prediction using object-oriented metrics," *J. Syst. Softw.*, vol. 76, no. 2, pp. 147–156, 2005.

[16] E. Paikari, M. M. Richter, and G. Ruhe, "Defect prediction using case-based reasoning: An attribute weighting technique based upon sensitivity analysis in neural networks," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 22, no. 6, pp. 747–768, 2012.

[17] V. Vashisht, M. Lal, and G. S. Sureshchanda, "A framework for software defect prediction using neural networks," *J. Softw. Eng. Appl.*, vol. 8, no. 8, pp. 384–394, 2015.

[18] K. Elish and M. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, vol. 81, no. 5, pp. 649–660, 2008.

[19] D. Gray, D. Bowes, and N. Davey, "Using the support vector machine as a classification method for software defect prediction with static code metrics," in *Proc. Int. Conf. Eng. Appl. Neural Netw.*, 2009, pp. 223–234.

[20] Z. Yan, X. Chen, and P. Guo, "Software defect prediction using fuzzy support vector regression," in *Proc. Int. Symp. Neural Netw.*, 2010, pp. 17–24.

[21] J. Wang, B. Shen, and Y. Chen, "Compressed C4. 5 models for software defect prediction," in *Proc. 12th Int. Conf. Qual. Softw.*, 2012, pp. 13–16.

[22] N. Seliya and T. M. Khoshgoftaar, "The use of decision trees for cost-sensitive classification: An empirical study in software quality prediction," *Data Mining Knowl. Discovery*, vol. 1, no. 5, pp. 448–459, 2011.

[23] T. Wang, and W. Li, "Naive Bayes software defect prediction model," in *Proc. Int. Conf. Comput. Intell. Softw. Eng.*, 2010, pp. 1–4.

[24] B. Turhan and A. B. Bener, "Software defect prediction: Heuristics for weighted Naïve Bayes," in *Proc. 2nd Int. Conf. Softw. Data Technol.*, 2007, pp. 244–249.

[25] S. Amasaki Y. Takagi, and O. Mizuno, "A Bayesian belief network for assessing the likelihood of fault content," in *Proc. 14th Int. Symp. Softw. Rel. Eng.*, 2003, pp. 215–226.

[26] N. Fenton, M. Neil, and W. Marsh, "On the effectiveness of early life cycle defect prediction with Bayesian nets," *Empirical Softw. Eng.*, vol. 13, no. 5, 2008, Art. no. 499.

[27] A. Okutan and O. T. Yıldız, "Software defect prediction using Bayesian networks," *Empirical Softw. Eng.*, vol. 19, no. 1, pp. 154–181, 2014.

[28] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah, "Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 6, pp. 534–550, Jun. 2018.

[29] N. V. Chawla *et al.*, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, 2002.

[30] T. M. Khoshgoftaar, K. Gao, and N. Seliya, "Attribute selection and imbalanced data: Problems in software defect prediction," in *Proc. 22nd IEEE Int. Conf. Tools Artif. Intell.*, 2010, vol. 1, pp. 137–144.

[31] M. M. Özturk and A. Zengin, "HSDD: A hybrid sampling strategy for class imbalance in defect prediction data sets," in *Proc. 11th Int. Conf. Digit. Inf. Manage.*, 2016, pp. 225–234.

[32] L. Chen *et al.*, "Tackling class overlap and imbalance problems in software defect prediction," *Softw. Qual. J.*, vol. 26, no. 1, pp. 97–125, 2018.

[33] A. Estabrooks, T. Jo, and N. Japkowicz, "A multiple resampling method for learning from imbalanced data sets," *Comput. Intell.*, vol. 20, no. 1, pp. 18–36, 2004.

[34] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Trans. Rel.*, vol. 62, no. 2, pp. 434–443, Jun. 2013.

[35] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Inf. Softw. Technol.*, vol. 58, pp. 388–402, 2015.

[36] Z. Sun, Q. Song, and X. Zhu, "Using coding-based ensemble learning to improve software defect prediction," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 42, no. 6, pp. 1806–1817, Nov. 2012.

[37] H. Wang, T. M. Khoshgoftaar, and A. Napolitano, "A comparative study of ensemble feature selection techniques for software defect prediction," in *Proc. 9th Int. Conf. Mach. Learn. Appl.*, 2010, pp. 135–140.

[38] X. Xia, D. Lo, and E. Shihab, "Elblocker: Predicting blocking bugs with ensemble imbalance learning," *Inf. Softw. Technol.*, vol. 61, pp. 93–106, 2015.

[39] M. J. Siers and M. Z. Islam, "Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem," *Inf. Syst.*, vol. 51, pp. 62–71, 2015.

[40] J. Zheng, "Cost-sensitive boosting neural networks for software defect prediction," *Expert Syst. Appl.*, vol. 37, no. 6, pp. 4537–4543, 2010.

[41] M. Liu, L. Miao, and D. Zhang, "Two-stage cost-sensitive learning for software defect prediction," *IEEE Trans. Rel.*, vol. 63, no. 2, pp. 676–686, Jun. 2014.

[42] X. Y. Jing, S. Ying, Z. W. Zhang, S. S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 414–423.

[43] D. Tomar and S. Agarwal, "Prediction of defective software modules using class imbalance learning," *Appl. Comput. Intell. Soft Comput.*, vol. 2016, 2016, Art. no. 6.

[44] X. Y. Jing, F. Wu, X. Dong, and B. Xu, "An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems," *IEEE Trans. Softw. Eng.*, vol. 43, no. 4, pp. 321–339, Apr. 2017.
[45] X. Yu *et al.*, "Cross-company defect prediction via semi-supervised clustering-based data filtering and MSTrA-based transfer learning," *Soft Comput.*, vol. 22, no. 10, pp. 3461–3472, 2018.
[46] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, Apr. 2005.
[47] A. Janes, M. Scotto, and W. Pedrycz, "Identification of defect-prone classes in telecommunication software systems using design metrics," *Inf. Sci.*, vol. 176, no. 24, pp. 3711–3734, 2006.
[48] T. M. Khoshgoftaar and K. Gao, "Count models for software quality estimation," *IEEE Trans. Rel.*, vol. 56, no. 2, pp. 212–222, Jun. 2007.
[49] K. Gao and T. M. Khoshgoftaar, "A comprehensive empirical study of count models for software fault prediction," *IEEE Trans. Rel.*, vol. 56, no. 2, pp. 223–236, Jun. 2007.
[50] W. Afzal, R. Torkar, and R. Feldt, "Prediction of fault count data using genetic programming," in *Proc. IEEE Int. Multitopic Conf.*, 2008, pp. 349–356.
[51] S. S. Rathore and S. Kuamr, "Comparative analysis of neural network and genetic programming for number of software faults prediction," in *Proc. Nat. Conf. Recent Adv. Electron. Comput. Eng.*, 2015, pp. 328–332.
[52] S. S. Rathore and S. Kumar, "Predicting number of faults in software system using genetic programming," *Procedia Comput. Sci.*, vol. 62, pp. 303–311, 2015.
[53] S. S. Rathore and S. Kumar, "A decision tree regression based approach for the number of software faults prediction," *ACM SIGSOFT Softw. Eng. Notes*, vol. 41, no. 1, pp. 1–6, 2016.
[54] M. Chen and Y. Ma, "An empirical study on predicting defect numbers," in *Proc. 28th Int. Conf. Softw. Eng. Knowl. Eng.*, 2015, pp. 397–402.
[55] S. S. Rathore and S. Kumar, "An empirical study of some software fault prediction techniques for the number of faults prediction," *Soft Comput.*, vol. 21, pp. 7417–7434, 2017.
[56] J. A. K. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural Process. Lett.*, vol. 9, no. 3, pp. 293–300, 1999.
[57] P. R. Srivastava and T. Kim, "Application of genetic algorithm in software testing," *Int. J. Softw. Eng. Appl.*, vol. 3, no. 4, pp. 87–96, 2009.
[58] S. J. Huang, N. H. Chiu, and L. W. Chen, "Integration of the grey relational analysis with genetic algorithm for software effort estimation," *Eur. J. Oper. Res.*, vol. 188, no. 3, pp. 898–909, 2008.
[59] A. S. Sayyad, T. Menzies, and H. Ammar, "On the value of user preferences in search-based software engineering: A case study in software product lines," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 492–501.
[60] J. Guo *et al.*, "A genetic algorithm for optimized feature selection with resource constraints in software product lines," *J. Syst. Softw.*, vol. 84, no. 12, pp. 2208–2221, 2011.
[61] S. Di Martino *et al.*, "A genetic algorithm to configure support vector machines for predicting fault-prone components," in *Proc. Int. Conf. Product Focused Softw. Process Improvement*, 2011, pp. 247–261.
[62] R. S. Wahono, N. S. Herman, and S. Ahmad, "Neural network parameter optimization based on genetic algorithm for software defect prediction," *Adv. Sci. Lett.*, vol. 20, no. 10/11, pp. 1951–1955, 2014.
[63] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "Hydra: Massively compositional model for cross-project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 42, no. 10, pp. 977–998, Oct. 2016.
[64] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," *Mach. Learn.*, vol. 3, no. 2, pp. 95–99, 1988.
[65] L. Torgo, P. Branco, and R. P. Ribeiro, "Resampling strategies for regression," *Expert Syst.*, vol. 32, no. 3, pp. 465–476, 2015.
[66] T. Menzies, R. Krishna, and D. Pryor, "The SEACRAFT repository of empirical software engineering data," 2017. [Online]. Available: https://zenodo.org/communities/seacraft
[67] T. Menzies, Z. Milton, B. Turhan, B. Cukic, and Y. J. A. Bener, "Defect prediction from static code features: Current results, limitations, new approaches," *Automated Softw. Eng.*, vol. 17, pp. 375–407, 2010.
[68] Y. Kamei *et al.*, "Revisiting common bug prediction findings using effort-aware models," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.
[69] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Eng.*, vol. 17, no. 4/5, pp. 531–577, 2012.
[70] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, 2010.
[71] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proc. Int. Conf. Predictor MODELS Softw. Eng.*, 2009, pp. 1–10.
[72] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for evaluating fault prediction models," *Empirical Softw. Eng.*, vol. 13, no. 5, pp. 561–595, 2008.
[73] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Comparing the effectiveness of several modeling methods for fault prediction," *Empirical Softw. Eng.*, vol. 15, no. 3, pp. 277–295, 2010.
[74] W. J. Long, J. L. Griffith, and H. P. Selker, "A comparison of logistic regression to decision-tree induction in a medical domain," *Comput. Biomed. Res.*, vol. 26, no. 1, pp. 74–97, 1993.
[75] H. D. Vinod, "A survey of ridge regression and related techniques for improvements over ordinary least squares," *Rev. Econ. Statist.*, vol. 60, pp. 121–131, 1978.
[76] G. A. F. Seber and A. J. Lee, *Linear regression analysis*, vol. 936. Hoboken, NJ, USA: Wiley, 2012.
[77] J. Xu *et al.*, "Cost-sensitive learning of SVM for ranking," in *Proc. Eur. Conf. Mach. Learn.*, 2006, pp. 833–840.
[78] C. Tantithamthavorn *et al.*, "Automated parameter optimization of classification techniques for defect prediction models," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 321–332.
[79] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bull.*, vol. 1, pp. 80–83, 1945.
[80] J. A. Ferreira and A. H. Zwinderman, "On the Benjamini–Hochberg method," *Ann. Statist.*, vol. 34, no. 4, pp. 1827–1849, 2006.
[81] V. B. Kampenes *et al.*, "A systematic review of effect size in software engineering experiments," *Inf. Softw. Technol.*, vol. 49, no. 11/12, pp. 1073–1086, 2007.
[82] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2013, pp. 45–54.
[83] G. Canfora, A. de Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Defect prediction as a multiobjective optimization problem," *Softw. Testing Verification Rel.*, vol. 25, no. 4, pp. 426–459, 2015.
[84] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 297–308.
[85] K. E. Bennin *et al.*, "The significant effects of data sampling approaches on software defect prioritization and classification," in *Proc. 11th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2017, pp. 364–373.
[86] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.
[87] H. Wang, T. M. Khoshgoftaar, and N. Seliya, "How many software metrics should be selected for defect prediction," in *Proc. 24th Int. Florida Artif. Intell. Res. Soc. Conf.*, 2011, pp. 69–74.
[88] T. M. Khoshgoftaar, K. Gao, and A. Napolitano, "An empirical study of feature ranking techniques for software quality prediction," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 22, no. 2, pp. 161–183, 2012.