Learning from Imbalanced Data for Predicting the Number of Software Defects

Xiao Yu[1,2], Jin Liu[1*], Zijiang Yang[3], Xiangyang Jia[1],Qi Ling[2],Sizhe Ye[1]
[1]State Key Lab. of Software Engineering, Computer School, Wuhan University, Wuhan, China
[2]Department of Computer Science, City University of Hong Kong, Kowloon Tong, China
[3]Department of Computer Science, Western Michigan University, Kalamazoo, Michigan, USA
*Corresponding author email: jinliu@whu.edu.cn

***Abstract*—Predicting the number of defects in software modules can be more helpful in the case of limited testing resources. The highly imbalanced distribution of the target variable values (i.e., the number of defects) degrades the performance of models for predicting the number of defects. As the first effort of an in-depth study, this paper explores the potential of using resampling techniques and ensemble learning techniques to learn from imbalanced defect data for predicting the number of defects. We study the use of two extended resampling strategies (i.e., SMOTE and RUS) for regression problem and an ensemble learning technique (i.e., the AdaBoost.R2 algorithm) to handle imbalanced defect data for predicting the number of defects. We refer to the extension of SMOTE and RUS for predicting the Number of Defects as SmoteND and RusND, respectively. Experimental results on 6 datasets with two performance measures show that these approaches are effective in handling imbalanced defect data. To further improve the performance of these approaches, we propose two novel hybrid resampling/boosting algorithms, called SmoteNDBoost and RusNDBoost, which introduce SmoteND and RusND into the AdaBoost.R2 algorithm, respectively. Experimental results show that SmoteNDBoost and RusNDBoost both outperform their individual components (i.e., SmoteND, RusND and AdaBoost.R2).**

Keywords—software defect prediction;data imbalance; resampling;ensemble learning

I. Introduction

Based on the investigation of software metrics [1-2] (also referred to as software features), software defect prediction utilizes historical defect data mined from software repositories to predict the defect-proneness of new software modules. Therefore, software defect prediction is often used to help to reasonably allocate limited testing resources [3-5]. So far, many efficient software defect prediction methods using statistical methods or machine learning techniques have been proposed [6-10], but they are usually confined to predicting a given software module being defective-prone or not by means of some binary classification techniques.

However, estimating the defect-proneness of a given set of software modules is not enough for software testing in practice due to plenty of criticisms of practicality, especially when there is a lack of testing resources [7, 11]. Now take a typical application scenario for example. A software development team develops a new software project, which contains 100 software modules. Due to the tight deadline, the test team can afford to inspect a small part of the project (e.g., only 20% software modules). A sound technical solution is to identify the modules that are most likely to be defective-prone before excuting unit tests. Therefore, the test team builds a model for predicting the defect-proneness of these modules or a model for predicting the number of defects in these modules using the historical defect data, including values of all software metrics and the number of defects. After extracting the same metrics from new software modules, the test team can use the learned models to classify these new modules defective-prone or not, or predict the number of defects in these new software modules. Assuming that the prediction result of the model for predicting the defect-proneness is that 30% of them may be defective-prone, since the test team only can inspect 20% new modules, they have no idea to which 20% of these modules should be inspected. But according to the prediction results of the model for predicting the number of defects, they can obtain an descending order of the 100 new modules based on the predicted number of defects, and allocate limited testing resources to discover the most numbers of defects according to the order (i.e., inspect the first 20% modules) [5]. Therefore, predicting the number of defects in software modules can be more helpful than predicting the modules being defective-prone or not in the case of limited testing resources [12].

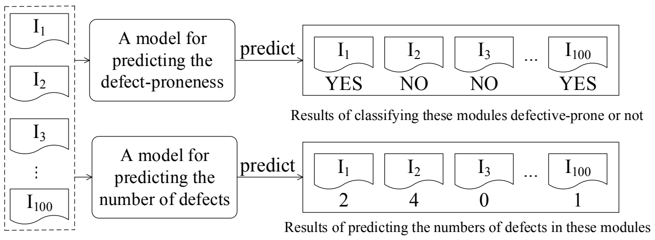
I1
I2
I3
I100
A model for predicting the defect-proneness
predict
I1 YES
I2 NO
I3 NO
I100 YES
Results of classifying these modules defective-prone or not
A model for predicting the number of defects
predict
I1 2
I2 4
I3 0
I100 1
Results of predicting the numbers of defects in these modules

Figure 1. An illustration of the difference between a model for predicting the defect-proneness and a model for predicting the number of defects.

A number of prior studies have investigated regression models for predicting the number of defects. Some researchers [13-16] have investigated genetic programming, decision tree regression, and multilayer perceptron for predicting the number of defects and found that these models achieved good performance. Chen et al. [17] performed an empirical study on predicting the number of defects using six regression algorithms and found that the prediction model built with decision tree regression had the highest prediction accuracy (i.e., the lowest root mean square error) in most cases. In

2332-6549/17 $31.00 © 2017 IEEE
DOI 10.1109/ISSRE.2017.18

another similar study, Rathore et al. [12] presented an experimental study to evaluate and compare the other six regression algorithms for predicting the number of defects. The results found that decision tree regression, Bayesian ridge regression, multilayer perceptron, and linear regression achieved better performance in terms of average absolute error (AEE) and average relative error (ARE). However, the highly imbalanced distribution of the target variable values (i.e., the number of defects) degrades the prediction performance. In most cases, the dataset contains much more non-defective modules than defective-prone ones. In other words, the number of defects in the majority of modules is zero, and the minority of modules have one or more defects. When regression models are trained by a highly skewed dataset, these models have weak capability to accurately predict the number of defects in a defective-prone module.

A common solution to forecasting tasks with imbalanced data is the use of resampling techniques [18], which balances the distribution by either adding examples to the minority class (oversampling) or removing examples from the majority class (under-sampling). Several resampling techniques for classification problem have been proposed, such as RUS (random under-sampling) and SMOTE (synthetic minority over-sampling technique) [19]. In addition to these resampling techniques, ensemble learning techniques have become another major category of approaches to handle imbalanced data, such as Bagging [20] and Boosting [21]. While resampling techniques manipulate training data to rectify the skewed distributions, ensemble learning techniques improve the performance by combining multiple weak prediction models (regardless of whether the data are imbalanced).

The aforementioned resampling techniques and ensemble learning techniques are confined to classification problem, i.e., predicting a given software module being defective-prone or not. Recently, efforts have been made to adapt resampling techniques and ensemble learning techniques to regression problem [22, 23]. Torgo et al. [23] adapted SMOTE and RUS for regression tasks, where the goal is to forecast rare extreme values of the target variable. Drucker et al. [24] proposed the Adaboost.R2 algorithm, which is a boosting algorithm for regression problem. However, it is still unclear what extent resampling techniques and ensemble learning techniques contribute to improving the performance of models for predicting the number of defects, and how to make better use of them to improve the performance of models for predicting the number of defects.

As the first effort of an in-depth study of resampling techniques and ensemble learning techniques for predicting the number of defects, this paper explores their potential by focusing on two research questions: Can resampling techniques and ensemble learning techniques be good solutions to predict the number of defects? Can we make better use of them? The answers will provide guidance and valuable information for choosing and designing good models for predicting the number of defects.

For the first question, we study the use of resampling techniques and ensemble learning techniques for predicting the number of defects. Our endeavor is based on three approaches: (i) the first is based on SMOTE; (ii) the second is based on RUS; (iii) the third is based on the Adaboost.R2 algorithm. The two resampling techniques were initially proposed for classification problem and were then extended for regression tasks [22]. We refer to the extension of SMOTE and RUS for predicting the Numbers of Defects as SmoteND and RusND, respectively. Using three regression models and two performance measures, we evaluate the performance of the three approaches using 6 publicly available project datasets. Experimental results show that the three approaches can be good solutions to learn from imbalanced data for predicting the number of defects.

For the second question, our objective is to develop a better solution that combines the strength of SmoteND, RusND and AdaBoost.R2. Inspired by the SMOTEBoost algorithm [58] and the RUSBoost algorithm [59], we present two novel hybrid resampling/boosting algorithms called SmoteNDBoost and RusNDBoost to learn from imbalanced data for predicting the number of defects. SmoteNDBoost introduces SmoteND into the Adaboost.R2 algorithm, while RusNDBoost embeds RusND in the Adaboost.R2 algorithm. We want to utilize SmoteND and RusND to balance the data distribution, and we want to employ the AdaBoost.R2 algorithm to improve the overall prediction performance using these balanced data. Experimental results show that both SmoteNDBoost and RusNDBoost achieve better performance than their individual components (i.e., SmoteND, RusND and AdaBoost.R2).

The remainder of this paper is organized as follows. Section II presents the related work. Section III introduces the preliminaries, i.e., the two resampling techniques and an ensemble learning technique studied in this work. Section IV and Section V show the experiment setup and experiment results, respectively. Section VI proposes two better solutions (i.e., SmoteNDBoost and RusNDBoost) to learn from imbalanced data for predicting the number of defects. Section VII discusses the potential threats to validity. Finally, Section VIII addresses the conclusion and points out the future work.

II. Related Work

In this section, we briefly review the existing defect prediction methods. These methods can be categorized into two main types: predicting the defect-proneness of software modules via classification techniques and predicting the number of defects in software modules via regression techniques.

A. Predicting the Defect-proneness of Software Modules

Support vector machine [25-27], neural networks [28-30], decision trees [31-32] and Bayesian methods [33-37] paved the way for classification-based methods in the flied of software defect prediction. These methods used software metrics to properly predict whether a module is defective-prone or not. However, the highly imbalanced nature of the defective-prone and non-defective classes of the data set degraded the prediction performance. Numerous methods have been proposed to cope with class imbalance problem. These methods can be categorized into four main types: resampling

[38-42], cost-sensitive [43-51], ensemble learning [52-57] and hybrid approaches [58-60].

Resampling techniques are classified as oversampling and under-sampling. Under-sampling reduces the number of instances in the majority class to balance the class distribution, whereas oversampling is a technique in which the minority class is over-sampled by creating synthetic instances. One of the most popular oversampling techniques is SMOTE [19], which generates synthetic instances based on a number of nearest neighbors. One of the most common under-sampling techniques is RUS, which simply selects a subset of majority class instances randomly and then combine them with minority class instances as a training set. Resampling techniques are simple and efficient, but their effectiveness depends greatly on the problem and training algorithms [42].

In the process of defect prediction, misclassify different software defect classes can be divided into two types, namely, "Type I" and "Type II" [44]. "Type I" misclassification cost and "Type II" misclassification cost are different. Some cost-sensitive learning methods [45-52] have been proposed to address the class imbalance problem by generating a classification model with minimum misclassification cost. The problem with cost-sensitive methods is the definition of the cost matrix as there is no systematic approach to do so.

In addition to the aforementioned resampling techniques and cost-sensitive methods, ensemble learning techniques [53-58] have become another major category of approaches to cope with imbalanced data. Boosting is one of the most popular ensemble learning techniques, which combines multiple weak learners to improve the performance. In particular, AdaBoost.M2 [21] is a popular boosting algorithm for classification problem. The set of training instances is assigned an equal weight at the beginning and the weight of instances is either increased or decreased depending on whether the weak classifiers of the current iteration classified that instance incorrectly or not. The next iterations focus on those instances with higher weights. In this way, AdaBoost.M2 builds a series of weak classifiers. Finally, the strong classifier is based on a weighted vote among these weak classifiers.

SMOTEBoost [58] and RUSBoost [59] are two most representative hybrid approaches to cope with class imbalance problem. SMOTEBoost is a combination of SMOTE and the AdaBoost.M2 algorithm, which outperforms both SMOTE and AdaBoost.M2. In SMOTEBoost, SMOTE is applied to the training data during each round of boosting to achieve a more balanced training data set. Similarly, RUSBoost is based on the AdaBoost.M2 algorithm, but it uses RUS instead of SMOTE.

In our paper, the proposed SmoteNDBoost algorithm is similar to SMOTEBoost. The differences between SmoteNDBoost and SMOTEBoost are as follows. SMOTEBoost combines SMOTE and the AdaBoost.M2 algorithm, while our proposed algorithm SmoteNDBoost combines SmoteND and the AdaBoost.R2 algorithm, which is a boosting algorithm for regression. Similarly, RusNDBoost combines RusND and the AdaBoost.R2 algorithm, while RUSBoost combines RUS and the Adaboost.M2 algorithm.

B. *Predicting the Number of Defects in Software Modules*

A number of prior studies have investigated some regression models for predicting the number of defects. Graves et al. [15] presented a generalized linear regression based method for predicting the number of defects using various change metrics datasets collected from a large telecommunication system and found that modules age, changes made to module and the age of the changes were significantly correlated with the defect-proneness. However, no performance measure was used to evaluate the appropriateness of generalized linear regression for predicting the number of defects [16]. Wang et al. [61] presented BugStates, a method for predicting the number of defects at each state based on defect state transition models. Ostrand et al. [62] and Yu et al. [63] employed negative binomial regression (NBR) model to predict the number of defects. They found that NBR is effective in predicting the number of defects. Janes et al. [64] used three count models (Poisson regression, NBR, and zero-inflated NBR) to predict the number of defects over five real-time telecommunication systems. The results found that zero-inflated NBR model achieved the best performance. Some researchers [12-15] used genetic programming (GP) to predict the number of defects and found that GP model produced significant predictive accuracy. Santosh et al. [16] explored the capability of decision tree regression (DTR) for predicting the number of defects in two different scenarios, intra-release prediction and inter-releases prediction for the given software system. The results showed that DTR model produced significant prediction accuracy for predicting the number of defects in both the considered scenarios.

Gao et al. [65-66] performed a comprehensive empirical study of five count models for predicting the number of defects. The study was performed over two industrial software systems. The results found that zero-inflated negative binomial regression and hurdle negative binomial regression models produced better prediction accuracy. Chen et al. [17] performed an empirical study of six regression algorithms for predicting the number of defects and found that the prediction model built with decision tree regression had the highest prediction accuracy in terms of root mean square error (RMSE). In another similar study, Rathore et al. [12] presented an empirical study to evaluate and compare the other six regression algorithms for predicting the number of defects. The results found that decision tree regression, genetic programming, Bayesian ridge regression, and linear regression achieved better performance in terms of ARE and AEE. However, the imbalanced distribution of the target variable values (i.e., the number of defects) degrades the predictive accuracy, but has not received much attention.

III. Preliminaries

In this section, we present SmoteND, RusND and AdaBoost.R2 to learn from imbalanced data for predicting the number of defects. Predicting the number of defects is a particular class of regression problem. In this context, given a software defect dataset $S=\{(\boldsymbol{x}_1,y_1), (\boldsymbol{x}_2,y_2),\ldots,(\boldsymbol{x}_n,y_n)\}$, where $\boldsymbol{x}_i$ is a feature vector representing the software metric values extracted from the *i*th instance, y_i is the target variable, i.e., the defect numbers of the *i*th instance, and *n* is the number of instances in *S*, our goal is to obtain a regression model $y=F(\boldsymbol{x})$.

A. SmoteDE

The SMOTE was initially proposed to address classification problem with imbalanced class distribution. Torgo et al. [22] proposed a variant of SMOTE for addressing regression problem where the key goal is to accurately predict rare extreme values, which they named SmoteR. There are three key issues of SmoteR in order to adapt SMOTE for regression problem: (i) how to define the normal target variable values and the rare target variable values; (ii) how to create new synthetic instances (i.e., over-sampling); and (iii) how to decide the target variable values of these new synthetic instances.

Regarding the first issue, SmoteR is based on a relevance function and on a user-specified threshold on the values of this function that leads to the definition of the rare target variable value. The instances with the rare target variable value are called as the rare instances, and the instances with the normal target variable value are called as the normal instances. For predicting the number of defects, we define the defective-prone modules as the rare instances and define non-defective modules as the normal instances. Regarding the second key issue, we use the same approach as in SMOTE and SmoteR to generate synthetic instances for predicting the number of defects. Finally, the third key issue is to decide the target variable value of the generated instances. In the original SMOTE algorithm, this is a trivial question, because all minority class instances have the same class, the same will happen to the instances generated from this set [22]. For regression task, the answer is not so trivial. The instances that are to be over-sampled do not have the same target variable value. This means that when a pair of instances are used to generate a new synthetic instance, they will not have the same target variable value. SmoteR uses 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. For predicting the number of defects, we use the same approach in SmoteR to decide the number of defects of the synthetic instance.

We refer to SmoteR for predicting the number of defects as SmoteND. Algorithm 1 presents the pseudo-code of SmoteND. If the number of the synthetic instances is less than the number of the original rare instances, we randomly select (*n*×*ratio*-*m*) rare instances to be used for generating new instances (Lines 1-3). Otherwise, $\frac{ratio\times(n-m)-m}{m}$ neighbors from the *k* nearest neighbors are randomly chosen (Line 6). In this paper, we choose *k* as 5. This setting is suggested by Chawla et al. [19]. For example, if we want to generate $2\times m$ rare instances, only two neighbors from the five nearest neighbors are chosen and one instance is generated in the direction of each. The key aspect of this algorithm is the generation of the synthetic instance. The feature vector of the synthetic instance is generated in the following way (Line 11): Take the difference of the feature vector between of the *i*th rare instance and its nearest neighbor. Multiply this difference by a random number between 0 and 1, and add it to the feature vector of the *i*th rare instance. The number of defects of the synthetic instance is a weighted average of the number of defects of the two seed instances (Lines 12-14). The weights are calculated as an inverse function of the distance of the generated instance to each of the two seed instances.

Algorithm 1. SmoteND

```
Input:  Defect dataset $S=\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2),\ldots,(\boldsymbol{x}_n, y_n)\}$
        Number of the rare instances, $m$
        Desired ratio between the rare instances and the normal instances, $ratio$
        Number of nearest neighbors, $k$
Output: Set $O$ of the synthetic rare instances
1.  if $ratio<2\times[m/(n-m)]$
2.      Randomize the $m$ rare instances;
3.      $m= ratio\times(n-m)-m$;
4.      $index=1$;
5.  else
6.      $index=$(int) ($\frac{ratio\times(n-m)-m}{m}$);
7.  for $i=1$ to $m$ do
8.      Calculate $k$ nearest neighbors for $i$ ;
9.      while $index \neq 0$
10.         Choose a random number between 1 and $k$, call it $nn$ ;
11.         $\boldsymbol{x}_{\text{synthetic}}=\boldsymbol{x}_i+\text{Random}(0,1)\times(\boldsymbol{x}_{nn}-\boldsymbol{x}_i)$;
12.         $d_1\leftarrow\text{DIST}(\boldsymbol{x}_{\text{synthetic}}, \boldsymbol{x}_i)$;
13.         $d_2\leftarrow\text{DIST}(\boldsymbol{x}_{\text{synthetic}}, \boldsymbol{x}_{nn})$;
14.         $y_{\text{synthetic}}=\frac{d_2\times y_i+d_1\times y_{nn}}{d_1+d_2}$;
15.         Add $(\boldsymbol{x}_{\text{synthetic}},y_{\text{synthetic}})$ to $O$;
16.         $index$--;
17.     end while
18. end for
19. return Set $O$ of the synthetic rare instances;
```

B. RusND

The RUS was initially proposed to address classification problem with imbalanced class distribution. The basic idea of RUS is to decrease the number of the normal instances to balance the ratio between the rare instances and the normal instances. Different from adapting SMOTE for regression problem, how to define the normal instances and the rare instances is the only key issue in order to adapt RUS for regression problem, because RUS does not involve generating new synthetic instances. Regarding the key issue, as we have mentioned in Section III-A, we define the defective-prone modules as the rare instances and define non-defective modules as the normal instances. We refer to RUS for predicting the number of defects as RusND.

The procedure of RusND is as follows:

(1) Determine the number p of selected normal instances according to the ratio between the rare instances and the normal instances;

(2) Randomly select p instances from the normal instances;

(3) Combine the selected normal instances and all the rare instances to obtain the training dataset.

Compared to SmoteND, the main drawback of RusND is the loss of information that comes with deleting instances from the training data [67]. It has the benefit, however, of decreasing the time required to train the regression model since the size of the training data set is reduced. On the other hand, SmoteND results in no lost information, but it increases model training times.

C. AdaBoost.R2

Ensemble learning combines a series of k weak learners with the aim of creating a composite prediction model to improve prediction accuracy. This paper uses AdaBoost.R2, which is a well-known boosting algorithm for regression problem. We present a brief description of the AdaBoost.R2 algorithm in this work due to the space limit. For the complete details of the AdaBoost.R2 algorithm, please refer to Harris Drucker's work [24].

Initially, AdaBoost.R2 assigns each instance from the training data set S an equal weight. Generating k weak regression models for the ensemble requires k rounds through the rest of the algorithm. In round i, the instances from S are sampled to form a training set, S_i, of size $|S|$. A weak regression model, M_i, is derived from the training instances of S_i. Next, a so-called loss function is introduced to compute the performance of the weak regression model using S as a test set. All the weights of the training instances are then updated according to the loss function. The process is repeated until a preset number of weak regression models are constructed or the average loss is less than 0.5. Finally, the output from different weak regression models will be combined to produce single prediction. The final output is the weighted median of the weak regression models' results.

In this paper, all boosting algorithms (Adaboost.R2, SmoteNDBoost, and RusNDBoost) are performed using fifty iterations. Preliminary experiments with the three algorithms using more iterations did not result in significant improvement.

IV. Experiment Setup

A. Data set

In this experiment, we employ 6 available and commonly used software project datasets with their 22 releases which can be obtained from PROMISE [67]. The details about the datasets is shown in Table I, where *#Instance* represents the number of instances in the release, *#Defects* represents the total number of defects in the release, *%Defect* represents the percentage of defective-prone instances in the release, *Max* is the maximum value of defects in the release, *Avg* is the average value of defects of all defective-prone instances in the release. There are the same 20 independent variables (i.e., the 20 software metrics) and one dependent variable (i.e., the number of defects) in the six datasets. For the complete details of the software metrics, please refer to [65-66].

TABLE I. Details of Experiment Dataset

Project	Release	#Instance	#Defects	%Defects	Max	Avg
Ant	1.3	125	33	16.0%	3	1.65
	1.4	178	47	22.5%	3	1.18
	1.5	293	35	10.9%	2	1.09
	1.6	351	184	26.2%	10	2.00
	1.7	745	338	22.3%	10	2.04
Camel	1.0	339	14	3.4%	2	1.08
	1.2	608	522	35.5%	28	2.42
	1.4	872	335	16.6%	17	2.31
	1.6	965	500	19.5%	28	2.66
Jedit	3.2	272	382	33.1%	45	4.24
	4.0	306	226	24.5%	23	3.01
	4.1	312	217	25.3%	17	2.75
	4.2	367	106	13.1%	10	2.21
	4.3	492	12	2.2%	2	1.09
Synapse	1.0	157	21	10.2%	4	1.31
	1.1	222	99	27.0%	7	1.65
	1.2	256	145	33.6%	9	1.69
Xalan	2.4	723	156	15.2%	7	1.42
	2.5	803	531	48.2%	9	1.37
	2.6	885	625	46.4%	6	1.52
Log4j	1.0	135	61	25.2%	9	1.79
	1.1	109	86	33.9%	9	2.32

B. Learners

This paper employs three regression models to predict the number of defects, Decision Tree Regression (DTR), Bayesian Ridge Regression (BRR), and Linear Regression (LR). The first reason we choose these regression models is that these models achieved better performance in most cases for predicting the number of defects [11-12]. The second reason we choose these regression models is that these models fall into three different families of learning methods. DTR is a decision-tree model [69]; BRR is a probabilistic model [68]; and LR is a statistical model [70].

It is worthy of note that we implement these three individual regression models based on the python machine learning library *sklearn*. We use the default parameter settings specified by *sklearn* for these models. That is, we do not perform additional optimizations for these models.

C. Performance measures

Previous studies [17], [65], [66] have employed some performance measures, such as average absolute error (AEE), average relative error (ARE), and root mean square error (RMSE) for evaluating the performance of models for predicting the number of defects. Using imbalanced defect data to derive a regression model and then estimate the error value of the resulting learned model can result in misleading over-optimistic estimates due to over-specialization of the learning algorithm to the imbalanced defect data. Suppose that we have trained a regression model to predict the number of defects in the project Ant 1.3, which contains 124 instances and 33 defects. An AEE value of, say, 0.264 (=33/124) may make the regression model seem quite accurate. But, an AEE value of

0.264 may not be acceptable—the regression model could predict the number of defects of all instances to be zero. Therefore, we need other performance measures.

Yang et al. [71] pointed out that predicting the precise number of defects of a module is hard to do due to the lack of good quality data in practice. Actually, for those existing approaches that tried to predict explicitly the number of defects in a software module, they used these predicted numbers to rank the modules anyway, to direct the software quality assurance team in targeting the most faulty modules first [15], [65], [72], [73]. In the beginning, the percentage of defects contained in the 20% of modules predicted to have the most faults was used to assess predictive accuracy [15]. However, the performance can be sensitive to the arbitrary cutoff value of 20%. Testing resources may be sufficient for testing the first 40% modules, or resources can test only the first 5% modules. Hence, Weyukers et al. [72] proposed fault-percentile-average (FPA) to reflect the effectiveness of the different prediction models across all values of the cutoff, and You et al. [73] employed Spearman's rank correlation coefficient and Kendall rank correlation coefficient [74] as the performance measure.

In the experiment, we employ Kendall rank correlation coefficient (Kendall for short) and FPA to measure the performance.

Kendall: Kendall rank correlation coefficient is a statistic used to measure the ordinal association between two measured quantities. Let (x_1, y_1), (x_2, y_2), …, (x_n, y_n) be a set of observations of the joint random variables X and Y respectively. In this paper, x_i and y_i are the actual number of defects and the predicted number of defect in ith instance, respectively. Any pair of observations (x_i, y_i) and (x_j, y_j), where i≠j, are said to be concordant if the ranks for both elements agree: that is, if both $x_i>x_j$ and $y_i>y_j$; or if both $x_i<x_j$ and $y_i<y_j$. They are said to be discordant, if $x_i>x_j$ and $y_i<y_j$; or if $x_i<x_j$ and $y_i>y_j$. If $x_i=x_j$ or $y_i=y_j$, the pair is neither concordant nor discordant. The Kendall τ coefficient is defined as:

$$\tau=\frac{(\text{number of concordant pairs})-(\text{number of discordant paris})}{n(n-1)/2}$$

FPA: Considering k modules listed in increasing order of predicted defect number as $f_1, f_2, f_3, \ldots, f_k$, and assuming that n_i is the actual defect number in the module i, $n=n_1+n_2+\ldots+n_k$ is the total number of defects, and the top predicted modules should have $\sum_{i=k-m+1}^{k} n_i$ defects. The proportion of the actual defects in the top m predicted modules to the whole defects is

$$\frac{1}{n}\sum_{i=k-m+1}^{k} n_i.$$

Then the FPA is define as

$$\frac{1}{k}\sum_{m=1}^{k}\frac{1}{n}\sum_{i=k-m+1}^{k} n_i.$$

FPA is actually the average of the proportions of actual defects in the top modules to the whole defects, which is a more comprehensive performance measure than the percentage of defects in the top 20% modules. A higher FPA means a better ranking, where the modules with most defects come first.

D. Experimental Design Summary

All experiments are performed using tenfold cross-validation. We merge the different releases of a project as a dataset and divide the dataset into ten folds of approximately equal size, nine of which are used to build the regression model, while the remaining partition is used to test the model. This cross-validation is repeated ten times so that each partitions are used exactly once as the test data. In this paper, we set the desired ratio between the rare instances and the normal instances as 100% for SmoteND and RusND. The above procedure is repeated 20 times in total to avoid sample bias. Overall performance measure for all approaches is estimated by averaging the results over 20 runs of tenfold cross-validation.

V. Experiment Results

In this section, we present the experiment results to answer the first research question mentioned in Section I. Table II records the average Kendall and FPA of all 6 datasets with four different approaches on three regression models DTR, BRR and LR. W/D/L (Kendall), short for Win/Draw/Loss (Kendall), presents the number of datasets, on which the approach in this column performs better than, the same as, or worse than None, in terms of Kendall. In the same way, W/D/L (FPA), short for Win/Draw/Loss (FPA), presents the number of datasets, on which the approach in this column performs better than, the same as, or worse than None, in terms of FPA. For example, the data of the four column of the four row is 5/0/1, it indicates that RusND outperforms None on 5 datasets and fails on 1 dataset in terms of FPA. For a more detailed description of the entire distribution of prediction performance across all datasets, Fig.1 and Fig.2 show the box-plots of Kendall and FPA values, with the four approaches for three regression models on the 6 datasets.

TABLE II. Average Performance of 6 Datasets with Three Regression Models on Kendall and FPA

Model	M	SmoteND	RusND	AdaBoost.R2	None
DTR	Kendall	0.345	0.307	**0.354**	0.232
	FPA	**0.718**	**0.718**	0.710	0.626
	W/D/L(Kendall)	6/0/0	6/0/0	6/0/0	
	W/D/L(FPA)	6/0/0	5/0/1	3/0/3	
BRR	Kendall	**0.321**	0.318	0.289	0.310
	FPA	**0.769**	0.768	0.746	0.758
	W/D/L(Kendall)	4/1/1	4/0/2	0/0/6	
	W/D/L(FPA)	6/0/0	6/0/0	0/0/6	
LR	Kendall	**0.323**	0.313	0.256	0.303
	FPA	**0.768**	0.763	0.725	0.754
	W/D/L(Kendall)	6/0/0	6/0/0	0/0/6	
	W/D/L(FPA)	6/0/0	6/0/0	0/0/6	

Our experimental results are in tune with the intuitive idea that resampling techniques and ensemble learning techniques improve the performance of models for predicting the number of defects. We can gain the following results from Table II and Figures 2-3.

(1) For DTR model, SmoteND and RusND achieves the best average FPA value, but fails in the best Kendall. But, the median value by RusND is higher than that by SmoteND. Regarding to the average Kendall, AdaBoost.R2 performs best.

The Win/Draw/Loss values show that, on three regression models, SmoteND, RusND, and AdaBoost.R2 outperform None on over half of datasets in terms of Kendall and FPA.

(2) For BRR model, AdaBoost.R2 is worse than None. That is, AdaBoost.R2 does not significantly improve the performance of the baseline learner and, in some cases, can hurt the performance. Despite this, SmoteND significantly improves the performance of this learner and achieves the best Kendall and FPA values. The Win/Draw/Loss values show that, on three regression models, SmoteND and RusND outperform None on over half of datasets in terms of Kendall and FPA.

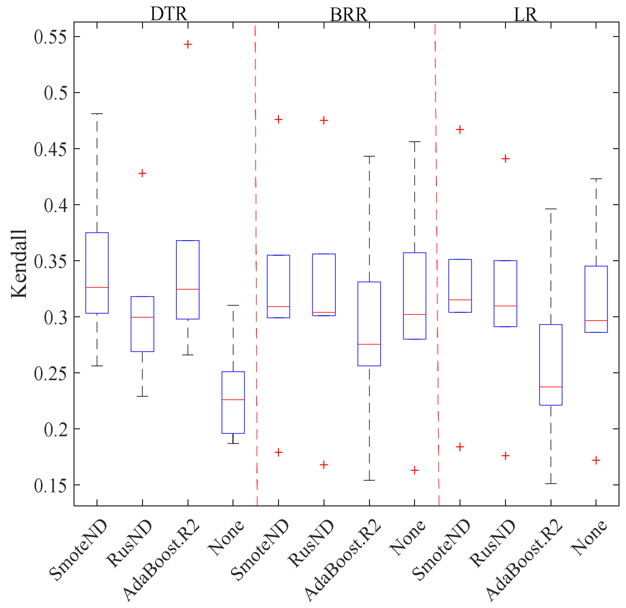

Figure 2. Box-plots for Kendall on 6 datasets with three regression models.

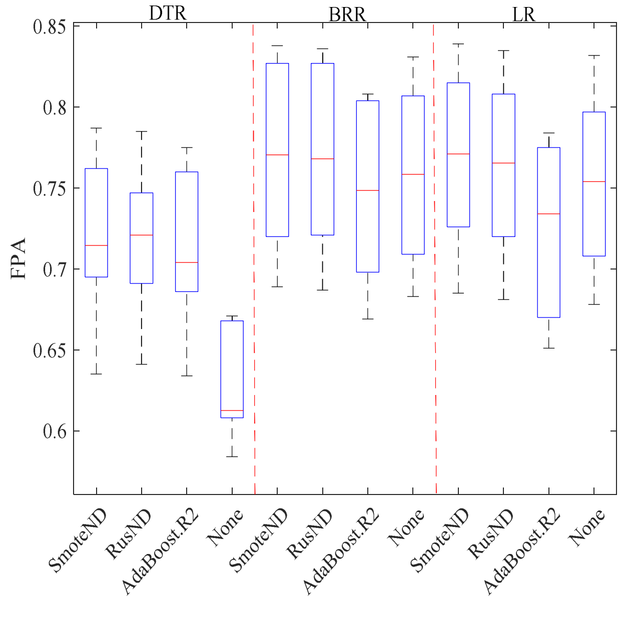

Figure 3. Box-plots for FPA on 6 datasets with three regression models.

(3) For LR model, SmoteND performs better FPA values than all the other approaches. The Win/Draw/Loss values show that, on three regression models, SmoteND and RusND outperform None on all datasets in terms of Kendall and FPA. But, AdaBoost.R2 is worse than None. That is, AdaBoost.R2 does not significantly improve the performance of the baseline learner and, in some cases, can hurt the performance.

To sum up, in almost all situations, SmoteND and RusND are significant than None. That is, the improvements obtained by resampling are not specific to any single learner or performance measures. With the exception of BRR and LR, AdaBoost.R2 generally performs similar to or better than None. Therefore, we can conclude that resampling techniques and ensemble learning techniques are good solutions to predict the number of defects.

VI. SMOTENDBOOST AND RUSNDBOOST

To further improve the performance of SmoteND, RusND and AdaBoost.R2, we propose SmoteNDBoost and RusNDBoost, which introduce SmoteND and RusND into the AdaBoost.R2 algorithm to learn from imbalanced data for predicting the number of defects.

A. SmoteNDBoost

Inspired by the SMOTEBoost algorithm [58], we propose a SmoteNDBoost algorithm that combines SmoteND and the AdaBoost.R2 algorithm. We want to utilize SmoteND to balance the data distribution, and we want to employ AdaBoost.R2 to improve the overall predictive performance using these balanced data.

Algorithm 2 presents the pseudo-code of SmoteNDBoost.

(1) In step 1, the weights of each instance are initialized to 1/*n*, where *n* is the number of instances in the training data set.

(2) In step 2, the average loss function $\overline{L_t}$ is initialized to 0.

(3) In step 3, *T* weak regression models are iteratively trained, as shown in steps 3a–3h. In step 3a, SmoteND is applied to create synthetic instances from rare instances until the new (temporary) training data set S_t' accord with the desired ratio between the rare instances and the normal instances. For example, if the desired ratio between the rare instances and the normal instances is 50:50, then the synthetic instances are created until the numbers of the rare instances and the normal instances are equal. As a result, S_t' will have a new weight distribution D_t'. Introducing SmoteND in each round of boosting will enable learner to learn from more of the rare instances. It is worthy to note that the synthetic instances are discarded after training a weak learner (step 3b) at iteration *t*. That is, they are not added to the original defect dataset. Therefore, we produce a different set of synthetic instances in each iteration, which increases the diversity amongst the regression models in the ensemble. After each boosting iteration, the error-estimation is on the original defect dataset *S* (steps 3c-3e). In step 3g, the weight update parameter β_t is calculated as $\overline{L_t}/(1-\overline{L_t})$. Next, the weight distribution for the next iteration D_{t+1} is updated (step 3h).

(4) After T iterations of step 3, the final prediction model $F(\boldsymbol{x})$ is returned as a weighted vote of the T weak learner (step 4).

```
Algorithm 2.  SmoteNDBoost
Input:  Defect dataset $S=\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2),\ldots,(\boldsymbol{x}_n, y_n)\}$
        Desired ratio between the rare instances and the normal instances, ratio
        Number of nearest neighbors, $k$
        Weak Learner, WeakLearn
        Number of iterations, $T$
Output: a prediction model $F(x)$
1. Initialize $D_1(i)=1/n$ for all $i$;
2. Initialize average loss function $\overline{L_t}=0$;
3. for $t=1$ to $T$ do
   (a) Create a temporary training dataset $S_t$' with distribution $D_t$' using SmoteND;
   (b) Train a weak learner $y=f_t(\boldsymbol{x})$ using $S_t$' and its distribution $D_t$';
   (c) Calculate the loss for each training instance in $S$ as $I_t(i)=|f_t(\boldsymbol{x}_i)-y_i|$;
   (d) Calculate the loss function $L_t(i)=\frac{I_t(i)}{Denom_t}$ for each training instance in $S$ where $Denom_t=\max_{i=1,2..n}(I_t(i))$;
   (e) Calculate the average loss $\overline{L_t}=\sum_{i=1}^{n} L_t(i)D_t(i)$;
   (g) Set $\beta_t=\overline{L_t}/(1-\overline{L_t})$;
   (h) Update distribution $D_t$ as $D_{t+1}(i)=\frac{D_t(i)\beta_t^{(1-L_t(i))}}{Z_t}$, where $Z_t$ is a normalization factor such that $D_{t+1}$ will be a distribution;
4. Output the final prediction model:
$F(\boldsymbol{x})=\inf\left[y \in R: \sum_{t:f_t(x)\le y}\log(\frac{1}{\beta_t}) \ge \frac{1}{2}\sum_t \log(\frac{1}{\beta_t})\right]$
```

B. RusNDBoost

RusNDBoost is based on the SmoteNDBoost algorithm, which is, in turn, based on the AdaBoost.R2 algorithm. SmoteNDBoost improves upon AdaBoost.R2 by introducing SmoteND, which helps to balance the distribution, while AdaBoost.R2 improves the overall predictive performance using these balanced data. The difference between RusNDBoost and SmoteNDBoost is that RusNDBoost applies RusND to the training data to achieve a more balanced training data set while SmoteNDBoost applies SmoteND. Therefore, Algorithm 2 can be modified to represent the RusNDBoost algorithm by changing step 3a to

```
Create a temporary training dataset $S_t$' with distribution $D_t$' using RusND;
```

The RusNDBoost algorithm can overcome two drawbacks of SmoteNDBoost. First, RusNDBoost decreases the complexity of the algorithm. SmoteND must find the k nearest neighbors of the rare instance and extrapolate between them to make new instances. On the other hand, RusND simply deletes the normal instances at random. Second, since SmoteND adds the synthetic instances to the training dataset, it results in longer model training times. The effect is compounded by SmoteNDBoost's use of boosting. On the other hand, RusND results in smaller training data sets and, therefore, shorter model training times of RusNDBoost.

C. Experimental Results

In this paper, we set the desired ratio between the rare instances and the normal instances as 100% for SmoteNDBoost and RusNDBoost. The ten-fold cross validation is repeated 20 times in total for SmoteNDBoost and RusNDBoost to avoid sample bias. Overall performance measures for SmoteNDBoost and RusNDBoost are estimated by averaging the results over 20 runs of tenfold cross-validation. Then, SmoteNDBoost and RusNDBoost are compared to their individual components (SmoteND, RusND and AdaBoost.R2) and None.

We perform the Wilcoxon signed-rank test [75] to analyze whether the performance values of SmoteNDBoost and RusNDBoost is statistically significant different with those of the compared approaches on three regression models over all datasets. The Wilcoxon signed-rank test is a non-parameter method of statistically significant test. For the performance values of two approaches compared, the null hypothesis is that there exists no significant difference between the two approaches. If the p-value that results from Wilcoxon test is less than 0.05, the null hypothesis is rejected. That is, the difference between the two approaches is identified as statistically significant. The significant test is implemented in IBM SPSS Statistics [76]. In additional, we compute the effect size, Hedges'g [77], to quantify the amount of difference between two approaches. A positive Hedges'g indicates that the performance of the prevision approach has a greater effect than that of the latter approach.

Tables III, IV, and V present the detailed FPA values of each datasets on three regression models with the p-values and Hedges'g values. The row labeled "p-value (1)" and the row labeled "Hedges'g (1)" present the comparison results between SmoteNDBoost and other approaches. The row labeled "p-value (2)" and the row labeled "Hedges'g (2)" present the comparison results between RusNDBoost and other approaches.

The following observations are derived from the data in Tables III, IV, and V.

(1) In almost all situations, both SmoteNDBoost and RusNDBoost perform significantly better than SmoteND, RusND, and AdaBoost.R2. In other words, the application of hybrid resampling/boosting is better than resampling and boosting alone, and the improvements obtained by hybrid resampling/boosting are not specific to any single learner.

(2) SmoteND and RusND are significantly better than AdaBoost.R2 and None on the three regression models.

(3) There is no significant difference between the performances of SmoteND and RusND. The average FPA values of SmoteND and RusND on the three regression models are very similar.

(4) While there is no significant difference between the performances of SmoteND and RusND, it is not the case that the performances between SmoteNDBoost and RusNDBoost are similar. With DTR and LR as the base learner, RusNDBoost outperforms SmoteNDBoost. With BRR as the base learner, SmoteNDBoost is preferred over RusNDBoost.

TABLE III. FPA VALUES ON 6 DATASETS USING DTR

Project	SmoteNDBoost	RusNDBoost	SmoteND	RusND	AdaBoost.R2	None
Ant	0.749	**0.757**	0.71	0.724	0.715	0.61
Camel	0.698	**0.725**	0.719	0.691	0.693	0.615
Jedit	0.760	**0.812**	0.787	0.785	0.76	0.671
Synapse	0.689	**0.718**	0.695	**0.718**	0.686	0.608
Xalan	**0.676**	0.659	0.635	0.641	0.634	0.584
Log4j	0.772	**0.775**	0.762	0.747	0.775	0.668
Avg	0.724	**0.741**	0.718	0.718	0.710	0.626
p-value(1)		0.116	0.600	0.596	0.104	**0.028**
Hedges'g(1)		-0.359	**0.126**	**0.140**	**0.289**	**2.557**
p-value(2)	0.116		**0.028**	**0.043**	0.420	0.270
Hedges'g(2)	**0.359**		**0.434**	**0.458**	**0.583**	**2.558**

TABLE IV. FPA VALUES ON 6 DATASETS USING BRR

Project	SmoteNDBoost	RusNDBoost	SmoteND	RusND	AdaBoost.R2	None
Ant	**0.828**	0.811	0.809	0.808	0.786	0.803
Camel	**0.776**	0.740	0.732	0.728	0.711	0.714
Jedit	**0.851**	0.821	0.838	0.836	0.804	0.831
Synapse	0.732	**0.753**	0.720	0.721	0.698	0.709
Xalan	**0.702**	0.699	0.689	0.687	0.669	0.683
Log4j	0.83	**0.840**	0.827	0.827	0.808	0.807
Avg	**0.787**	0.777	0.769	0.768	0.746	0.758
p-value(1)		0.345	**0.027**	**0.027**	**0.028**	**0.027**
Hebdges'g(1)		**0.159**	**0.282**	**0.303**	**0.673**	**0.467**
p-value(2)	0.345		0.249	0.248	**0.028**	**0.075**
Hedges'g(2)	-0.159		**0.138**	**0.160**	**0.543**	**0.331**

TABLE V. FPA VALUES ON 6 DATASETS USING LR

Project	SmoteNDBoost	RusNDBoost	SmoteND	RusND	AdaBoost.R2	None
Ant	0.819	**0.826**	0.806	0.803	0.764	0.797
Camel	0.736	**0.759**	0.736	0.728	0.704	0.714
Jedit	0.841	**0.853**	0.839	0.835	0.784	0.832
Synapse	**0.747**	**0.747**	0.726	0.72	0.670	0.708
Xalan	0.701	**0.716**	0.685	0.681	0.651	0.678
Log4j	0.819	**0.821**	0.815	0.808	0.775	0.794
Avg	0.777	**0.787**	0.768	0.763	0.725	0.754
p-value(1)		**0.043**	**0.043**	**0.028**	**0.028**	**0.027**
Hebdges'g(1)		-0.178	**0.159**	**0.250**	**1.230**	**0.394**
p-value(2)	**0.043**		**0.028**	**0.028**	**0.028**	**0.028**
Hedges'g(2)	**0.178**		**0.335**	**0.426**	**1.450**	**0.573**

Figure 4 shows the box-plot of Kendall values, with the six approaches for three regression models on the 6 datasets. We can gain the following results from Figure 4.

(1) For DTR model, the median values by SmoteNDBoost and RusNDBoost are higher than that by SmoteND, RusND, AdaBoost.R2 and None. In addition, the maximum value by SmoteNDBoost is much higher than that by other approaches, except AdaBoost.R2.

(2) For BRR model, the median value by SmoteNDBoost is much higher than that by other approaches, while the median value by RusNDBoost is a little lower than that by SmoteND. In addition, the maximum values by SmoteNDBoost and RusNDBoost is higher than that by AdaBoost.R2 and None and a little lower than that by SmoteND and RusND.

(3) For LR model, the median value by RusNDBoost is much higher than that by all other approaches, and the median value by SmoteNDBoost is a little lower than AdaBoost.R2. The maximum values by SmoteNDBoost and RusNDBoost are higher than that by other approaches, except AdaBoost.R2 and SmoteND.

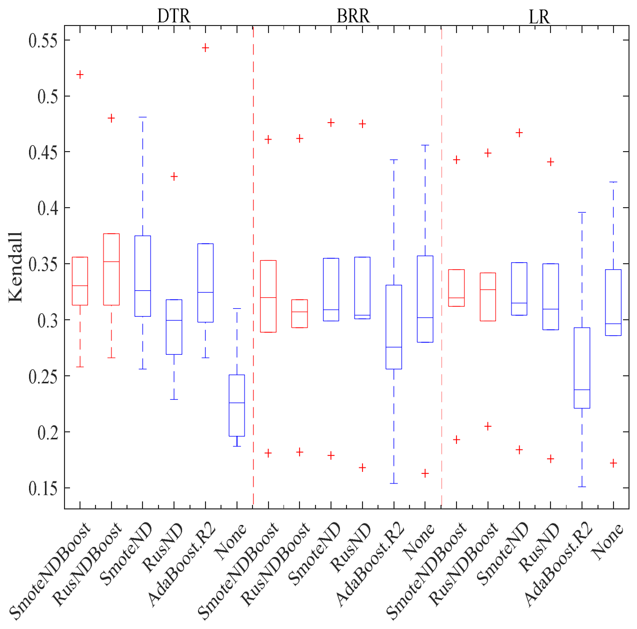

Figure 4. Box-plots for Kendall on 6 datasets with three regression models.

Finally, we directly compare SmoteNDBoost and RusNDBoost in Table VI. We revert to computing a standard t-statistic [78] to compare the means of these two approaches to obtain a more precise comparison. Table VI compares only SmoteNDBoost and RusNDBoost, with a two-sample t-statistic calculated for each learner and data set, presented by a performance metric. Therefore, each column totals to 18 (3 learners × 6 datasets), and in total, 36 pairwise comparisons between SmoteNDBoost and RusNDBoost were performed, each with a 95% confidence level. The first row represents the number of times that SmoteNDBoost significantly outperforms RusNDBoost, the second row is the number of times that RusNDBoost significantly outperforms SmoteNDBoost, and the final row represents the cases with no significant difference between SmoteNDBoost and RusNDBoost. Overall, SmoteNDBoost is comparably to RusNDBoost, particularly relative to the Kendall and FPA performance measure. However, SmoteND's main drawback (increasing the model training time due to larger training data sets) is amplified by its combination with AdaBoost.R2. Given the similar performance between SmoteNDBoost and RusNDBoost, one would prefer the simpler and faster approach: RusNDBoost.

TABLE VI. t-Test Comparison of SmoteNDBoost and RusNDBoost

	Kendall	FPA	Total
SmoteNDBoost	3	4	7
RusNDBoost	4	5	9
Neither	11	9	20

To sum up, in almost all situations, both SmoteNDBoost and RusNDBoost perform significantly better than SmoteND, RusND, AdaBoost.R2 and None, and RusNDBoost performs comparably to SmoteNDBoost while being a simpler and faster approach.

VII. Threats to Validity

In this section, we discuss several validity threats that may have an impact on the results of our studies.

External validity. Threats to external validity occur when the results of our experiments cannot be generalized. Although these datasets have been widely used in many software defect prediction studies, we still cannot claim that our conclusion can be generalized to other datasets. Another threat is the choice of the desired ratio between the rare instances and the normal instances. In this paper, we choose 5 different desired ratios (i.e., 80%, 90%, 100%, 110%, and 120%). For different datasets, the best desired ratio might be different, which might lead to different results.

Internal validity. We list several concerns about the bias in regression models selection and the incorrect implementation process of experiments. To avoid these threats, we choose three state-of-the-art regression models, which represent three categories: BRR as a probabilistic model, DTR a decision-tree model, LR as a statistical model. For the implementation, we use the python machine learning library *sklearn* to avoid the potential faults during the implementation process of the experiment.

Construct validity. Threats to construct validity focus on the bias of the measures used to evaluate the prediction performance. In our experiments, we employ Kendall rank correlation coefficient and FPA as the evaluation measures. Nonetheless, other evaluation measures such as Spearman's rank correlation coefficient and cost effectiveness graph [79] can also be considered.

Conclusion validity. Threats to conclusion validity focus on the statistical analysis method. In this work, we use Wilcoxon signed-rank test to statistically analyze the six approaches and a standard t-statistic test to compare SmoteNDBoost and RusNDBoost.

VIII. Conclusion and Future Work

Predicting the number of defects in software modules can be more helpful instead of predicting the modules being defective-prone or not. The imbalanced distribution of the target variable values (i.e., the number of defects) is the main cause of its learning difficulty, but has not received much attention. As the first effort of an in-depth study, this paper studies whether and how resampling techniques and ensemble learning techniques can improve the performance of models for predicting the number of defects. We investigate two extended resampling techniques (i.e., SMOTE and RUS) for regression tasks and an ensemble learning technique (i.e., AdaBoost.R2) in comparison with three top-ranked regression models (DTR, BRR, and LR). Experiments on 6 widely-studied project datasets with two performance measures indicate that resampling techniques and ensemble learning techniques can contribute to improving the performance of models for predicting the number of defects.

To further improve the prediction performance, we propose two novel hybrid resampling/boosting algorithms called SmoteNDBoost and RusNDBoost, to alleviate the problem of imbalanced data distribution for predicting the number of defects. We evaluate SmoteNDBoost and RusNDBoost, as well as their individual components (SmoteND, RusND and AdaBoost.R2). Experimental results show that both SmoteNDBoost and RusNDBoost perform significantly better than SmoteND, RusND, and AdaBoost.R2. RusNDBoost performs comparably to SmoteNDBoost, while being a simpler and faster approach.

Further work from this paper includes the investigation of other resampling techniques and ensemble learning techniques. Currently, this paper only considers SMOTE, RUS, and AdaBoost.R2. In addition, as mentioned in Section I, a test team can allocate limited testing resources according to the order of new modules based on the predicted numbers of defects, which ignores the module size, testing cost, and severity of defects. Some applications might prefer allocating limited testing resources according to the severity of defects, or require the test team to consider testing cost. This will be one of our future research interests. Moreover, we plan to apply our method to a real-life application [80-81].

Acknowledgment

The authors would like to acknowledge the support provided by the grands of the National Natural Science Foundation of China (61572374, No.U163620068, U1135005), and Academic Team Building Plan for Young Scholars from Wuhan University (WHU2016012).

References

[1] F. Rahman, D. Posnett, and P. Devanbu. Recalling the imprecision of cross-project defect prediction. *In Proceedings of the ACM SIGSOFT*

20th International Symposium on the Foundations of Software Engineering, 61,2012.

[2] K. Gao, T.M. Khoshgoftaar, and H. Wang. Choosing software metrics for defect prediction: an investigation on feature selection techniques. Software Practice & Experience,41(5):579-606, 2011.

[3] M. Shepperd, D. Bowes, and T. Hall. Researcher Bias: The Use of Machine Learning in Software Defect Prediction. *IEEE Transactions on Software Engineering*,40(6):603-616, 2014.

[4] Q. Song, Z. Jia, and M. Shepperd. A general software defect proneness prediction framework, Software Engineering. *IEEE Transactions on Software Engineering*, 37(3): 356-370, 2011.

[5] X. Yang, K. Tang, and X. Yao. A Learning-to-Rank Approach to Software Defect Prediction. *IEEE Transactions on Reliability*,64(1): 234-246, 2015.

[6] C. Catal. Software fault prediction: A literature review and current trends. Expert systems with applications, 38(4): 4626-4636, 2011.

[7] R. Malhotra. A systematic review of machine learning techniques for software fault prediction, Applied Soft Computing, 27: 504-518, 2015.

[8] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6): 603-616, 2014.

[9] Guo L, Ma Y, Cukic B, et al. Robust prediction of fault-proneness by random forests. *In 15th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2004: 417-428.

[10] Lu H, Kocaguneli E, Cukic B. Defect prediction between software versions with active learning and dimensionality reduction. *In 25th International Symposium on Software Reliability Engineering (ISSRE),*. IEEE, 2014: 312-322.

[11] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on software engineering*, 25(5): 675-689, 1999.

[12] S. S. Rathore and S. Kumar. An empirical study of some software fault prediction techniques for the number of faults prediction. Soft Computing, 1-18, 2016.

[13] Rathore S S and Kuamr S. Comparative analysis of neural network and genetic programming for number of software faults prediction. *National Conference on Recent Advances in Electronics & Computer Engineering (RAECE)*, 328-332, 2015.

[14] W. Afzal, R. Torkar, and R.Feldt. Prediction of fault count data using genetic programming. *Multitopic Conference, 2008. INMIC 2008*. IEEE International. IEEE, 2008.

[15] S. S. Rathore and S. Kumar. Predicting number of faults in software system using genetic programming. *Procedia Computer Science*, 62: 303-311, 2015.

[16] S. S. Rathore and S.Kumar. A Decision Tree Regression based Approach for the Number of Software Faults Prediction. *ACM SIGSOFT Software Engineering Notes*, 41(1): 1-6, 2016.

[17] M. Chen and Y. Ma. An empirical study on predicting defect numbers. In *Proceedings of the 28th International Conference on Software Engineering and Knowledge Engineering*, 397-402, 2015.

[18] Wang, Shuo, and X. Yao. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2): 434-443, 2013.

[19] Chawla and V. Nitesh. SMOTE: synthetic minority over-sampling technique. The Journal of artificial intelligence research, 16: 321-357, 2002.

[20] L. Breiman. Bagging predictors. *Machine learning*, 24(2): 123-140, 1996.

[21] Freund, Yoav, and E. Robert. Schapire. Experiments with a new boosting algorithm. icml, 96: 148-156, 1996.

[22] L. Torgo, P. Branco, and R. P. Ribeiro. Resampling strategies for regression. Expert Systems, 32(3): 465-476, 2015.

[23] L. Torgo, R. P. Ribeiro, and B. Pfahringer. Smote for regression. Portuguese conference on artificial intelligence. Springer Berlin Heidelberg, 378-389, 2013.

[24] Drucker and Harris. Improving regressors using boosting techniques. *ICML*. Vol. 97, 1997.

[25] K. Elish and M. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649-660, 2008.

[26] D. Gray, D. Bowes, and N. Davey. Using the support vector machine as a classification method for software defect prediction with static code metrics. *International Conference on Engineering Applications of Neural Networks*. Springer Berlin Heidelberg, 223-234, 2009.

[27] Z. Yan, X. Chen, and P. Guo. Software defect prediction using fuzzy support vector regression. *International Symposium on Neural Networks*. Springer Berlin Heidelberg, 17-24, 2010.

[28] M. M. T. Thwin and T. S.Quah. Application of neural networks for software quality prediction using object-oriented metrics. *Journal of systems and software*, 76(2): 147-156, 2005.

[29] 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. *International Journal of Software Engineering and Knowledge Engineering*, 22(06): 747-768, 2012.

[30] V. Vashisht, M. Lal, and G. S. Sureshchanda. A framework for software defect prediction using neural networks. *Journal of Software Engineering and Applications*, 8(8): 384, 2015.

[31] J. Wang, B. Shen, and Y.Chen. Compressed C4. 5 models for software defect prediction. *In Proceedings of the 12th International Conference on Quality Software*. IEEE, 13-16, 2012.

[32] N. Seliya and T. M.Khoshgoftaar. The use of decision trees for cost‐sensitive classification: an empirical study in software quality prediction. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 1(5): 448-459, 2011.

[33] T. Wang and W. Li. Naive bayes software defect prediction model. *Computational Intelligence and Software Engineering(CISE)*. IEEE, 1-4, 2010.

[34] B. Turhan and A. B. Bener. Software Defect Prediction: Heuristics for Weighted Naïve Bayes. *ICSOFT (SE)*, 244-249, 2007.

[35] S. Amasaki and Y. Takagi, Mizuno O. A bayesian belief network for assessing the likelihood of fault content. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 215-226, 2003.

[36] Fenton N, Neil M, and Marsh W. On the effectiveness of early life cycle defect prediction with Bayesian Nets. *Empirical Software Engineering*, 13(5): 499, 2008.

[37] Okutan, Ahmet, and O. T. Yıldız. Software defect prediction using Bayesian networks. *Empirical Software Engineering*, 19(1): 154-181, 2014.

[38] Khoshgoftaar, M. Taghi, K. Gao, and N. Seliya. Attribute selection and imbalanced data: Problems in software defect prediction. In *Proceedings of the 22th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE International Conference on. Vol. 1. IEEE, 2010.

[39] Ozturk, M. Maruf, and A. Zengin. HSDD: a hybrid sampling strategy for class imbalance in defect prediction data sets. In *Proceedings of the 5th Future Generation Communication Technologies (FGCT)*. IEEE, 2016.

[40] Pelayo, Lourdes, and S. Dick. Applying novel resampling strategies to software defect prediction. *Fuzzy Information Processing Society*, 2007. NAFIPS'07. *Annual Meeting of the North American*. IEEE, 2007.

[41] L. Chen, B. Fang, and Z. Shang. Tackling class overlap and imbalance problems in software defect prediction. *Software Quality Journal*, 1-29, 2016.

[42] Estabrooks, Andrew, T. Jo, and N. Japkowicz. A multiple resampling method for learning from imbalanced data sets. *Computational intelligence*, 20(1): 18-36, 2004.

[43] Zhou, Z. Hua, and X. Y. Liu. Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Transactions on Knowledge and Data Engineering,* 18(1): 63-77, 2006.

[44] Ting and K. Ming. An instance-weighting method to induce cost-sensitive trees. *IEEE Transactions on Knowledge and Data Engineering*, 14(3): 659-665, 2002.

[45] T. M. Khoshgoftaar, E. Geleyn and L. Nguyen. Cost-sensitive boosting in software quality modeling, *High Assurance Systems Engineering*, 2002. *7th IEEE International Symposium on*. IEEE, 51-60, 2002.

[46] J. Zheng. Cost-sensitive boosting neural networks for software defect prediction, *Expert Systems with Applications*, 37(6):4537-4543, 2010.

[47] M. Liu, L. Miao, and D. Zhang. Two-stage cost-sensitive learning for software defect prediction. *IEEE Transactions on Reliability*, 63(2):676-686, 2014.

[48] X. Y. Jing, S. Ying, Z. W. Zhang, S. S. Wu, and J. Liu. Dictionary learning based software defect prediction. *In Proceedings of the 36th International Conference on Software Engineering (ICSE).*ACM, 414-423, 2014.

[49] I. H. Laradji, M. Alshayeb, and L. Ghouti. Software defect prediction using ensemble learning on selected features. *Information and Software Technology*, 58: 388-402, 2015.

[50] 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. *Information Systems*, 51: 62-71, 2015.

[51] T., Divya, and S. Agarwal. Prediction of defective software modules using class imbalance learning. *Applied Computational Intelligence and Soft Computing*, 2016: 6, 2016.

[52] Wang, Shuo, H. Chen, and X. Yao. Negative correlation learning for classification ensembles. Neural Networks (IJCNN), *The 2010 International Joint Conference on*. IEEE, 2010.

[53] Wang, X. Benjamin, and N. Japkowicz. Boosting support vector machines for imbalanced data sets. *Knowledge and information systems*, 25(1): 1-20, 2010.

[54] Z. Sun, Q. Song, and X. Zhu. Using coding-based ensemble learning to improve software defect prediction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6): 1806-1817, 2012.

[55] Wang, Huanjing, T. M. Khoshgoftaar, and A. Napolitano. A comparative study of ensemble feature selection techniques for software defect prediction. *Machine Learning and Applications (ICMLA)*, *2010 Ninth International Conference on. IEEE*, 2010.

[56] X. Xia, D. Lo, and E. Shihab. Elblocker: Predicting blocking bugs with ensemble imbalance learning, *Information and Software Technology*, 61: 93-106, 2015.

[57] T. L. Graves, A. F. Karr, and J. S. Marron. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7): 653-661, 2000.

[58] N. V. Chawla, A. Lazarevic, and L. O. Hall. SMOTEBoost: Improving prediction of the minority class in boosting. *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer Berlin Heidelberg, 107-119, 2003.

[59] C. Seiffert, T. M. Khoshgoftaar, and J. Van Hulse. RUSBoost: A hybrid approach to alleviating class imbalance. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 40(1): 185-197, 2010.

[60] Domingos and Pedro. Metacost: A general method for making classifiers cost-sensitive. *In Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 1999.

[61] Wang, Jue, and H. Zhang. Predicting defect numbers based on defect state transition models. *Empirical Software Engineering and Measurement (ESEM), 2012 ACM-IEEE International Symposium on*. IEEE, 2012.

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

[63] Yu L. Using negative binomial regression analysis to predict software faults: a study of apache ant, 2012.

[64] A. Janes, M. Scotto, and W. Pedrycz. Identification of defect-prone classes in telecommunication software systems using design metrics, *Information sciences*, 176(24): 3711-3734, 2006.

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

[66] T. M. Khoshgoftaar and K. Gao. Count models for software quality estimation. *IEEE Transactions on Reliability*, 56(2): 212-222, 2007.

[67] G. Boetticher, T. Menzies and T. Ostrand, The PROMISE Repository of Empirical Software Engineering Data, <http://promisedata.org/repository>, 2007.

[68] H. D. Vinod. A survey of ridge regression and related techniques for improvements over ordinary least squares. *The Review of Economics and Statistics*, 121-131, 1978.

[69] W. J. Long, J. L. Griffith, and Selker H P. A comparison of logistic regression to decision-tree induction in a medical domain. *Computers and Biomedical Research*, 26(1): 74-97, 1993.

[70] Seber, George AF, and Alan J. Lee. Linear regression analysis. Vol. 936. John Wiley & Sons, 2012.

[71] Yang, Xiaoxing, K. Tang, and X. Yao. A learning-to-rank approach to software defect prediction. *IEEE Transactions on Reliability,* 64(1): 234-246, 2015.

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

[73] You, Guoan and Y. Ma. A Ranking-Oriented Approach to Cross-Project Software Defect Prediction: An Empirical Study.

[74] M. G. Kendall. A new measure of rank correlation. Biometrika, 30(1/2): 81-93, 1938.

[75] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 80-83, 1945.

[76] A.P. Field. Discovering statistics using SPSS for Windows: Advanced techniques for the beginner. Discovering Statistics Using SPSS for Windows: Advanced Techniques for Beginners. Sage Publications, Inc., 2000.

[77] Kampenes, V. By, et al, A systematic review of effect size in software engineering experiments, *Inform. Softw. Technol*. 49.11 (2007) 1073-1086.

[78] Winer B J, Brown D R, Michels K M. Statistical principles in experimental design. New York: McGraw-Hill, 1971.

[79] Jiang T, Tan L, Kim S. Personalized defect prediction. *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. IEEE*, 2013.

[80] Liu Z, Wei C, Ma Y, et al. UCOR: an unequally clustering-based hierarchical opportunistic routing protocol for WSNs. *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, Berlin, Heidelberg, 2013: 175-185.

[81] Liu Z, Niu X, Lin X, et al. A Task-Centric Cooperative Sensing Scheme for Mobile Crowdsourcing Systems. *Sensors*, 2016, 16(5): 746.