International Journal of Software Engineering
and Knowledge Engineering
Vol. 28, No. 6 (2018) 811–830

DOI: 10.1142/S0218194018500237

An Empirical Study on Software Defect Prediction Using Over-Sampling by SMOTE

Cholmyong Pak*,†,‡, Tian Tian Wang*,§ and Xiao Hong Su*,¶

*Harbin Institute of Technology
Harbin 150001, P. R. China

†Kim Il Sung University, Pyongyang
Democratic People's Republic of Korea
‡pakcholmyong@yahoo.com
§sweetwtt@126.com
¶sxh@hit.edu.cn

Received 13 February 2017
Accepted 10 November 2017

Software defect prediction suffers from the class-imbalance. Solving the class-imbalance is more important for improving the prediction performance. SMOTE is a useful over-sampling method which solves the class-imbalance. In this paper, we study about some problems that faced in software defect prediction using SMOTE algorithm. We perform experiments for investigating how they, the percentage of appended minority class and the number of nearest neighbors, influence the prediction performance, and compare the performance of classifiers. We use paired t-test to test the statistical significance of results. Also, we introduce the effectiveness and ineffectiveness of over-sampling, and evaluation criteria for evaluating if an over-sampling is effective or not. We use those concepts to evaluate the results in accordance with the evaluation criteria for the effectiveness of over-sampling.

The results show that they, the percentage of appended minority class and the number of nearest neighbors, influence the prediction performance, and show that the over-sampling by SMOTE is effective in several classifiers.

Keywords: Software defect prediction; class-imbalance; fault prediction; over-sampling; SMOTE.

1. General Appearance

Software defect prediction is an important research field of software engineering and helps developers to improve the quality of software and reduce the cost of software. As the size and complexity of software grow, software defect prediction is getting more important. Software defect is cost-expensive, so it is crucial for developers to find the defects before releasing the software.

Typical software defect prediction is to predict whether there is a defect in software or not on the basis of software metrics data collected from previously developed products. Thus, software defect prediction can be considered as a learning problem over the software metrics datasets. Software metrics datasets are collections of software metrics which are quantitative measures to reflect the complexity of software such as physical size and control structure. These software metrics datasets commonly include Line of Code (LOC), Cyclomatic Complexity Metrics (CCM) [25], Halstead Complexity Metrics (HCM) [26] and Object Oriented Metrics (OOM) [27].

In software defect prediction, the class of interest is the defective module, but defective modules are much less than nondefective modules in historical data, that is, software metrics datasets are very class-imbalanced. (A dataset is class-imbalanced if the class of interest is much less than the other class [24].) Since software defect prediction is based on software metrics datasets, it cannot avoid the class-imbalance problem. Class-imbalance affects the accuracy of defect prediction and enables the predictor not find the defective-modules well. Therefore, it is necessary for defect prediction to solve the class-imbalance problem.

In general, the class-imbalance problem is a common problem in classification. With the development of data mining technologies, there were many research works which attempted to solve the class-imbalance problem in software defect prediction. Such research was performed in three ways [12, 13].

Firstly, some researchers tried to use cost-sensitive learning to solve the class-imbalance problem. Zhou and Liu applied cost-sensitive neural networks to software defect prediction and defined which methods are effective in training cost-sensitive neural networks, and concluded that threshold-moving was appropriate for training cost-sensitive neural networks [6]. Siers and Islam proposed a method using a cost sensitive decision forest and voting to solve the class-imbalance problem in software defect prediction [1]. Arara and Ayanba used the cost-sensitive neural network to set the costs of misclassifying the positive and negative classes with relevant coefficients, and improved the performance of prediction [3]. Zheng used three cost-sensitive algorithms which enable predictor to predict the defective modules well and compared the effectiveness of boosting neural networks for software defect prediction [7]. Moser *et al.* applied cost-sensitive learning to defect prediction and used three classifiers to analyze the effectiveness of prediction, and showed that the process metrics were significant for defect prediction [14].

Secondly, some researchers tried to use sampling technologies such as random over-sampling, under-sampling and SMOTE to solve the class-imbalance problem. Estabrooks *et al.* studied about the effectiveness over-sampling and under-sampling technologies for class-imbalanced data and found that the combination of two technologies was better than individual technologies [21]. Chawla *et al.* proposed a SMOTE algorithm which used original instances of minority class to synthesize new samples of minority class and then append them to datasets [11]. Shatnawi proposed a duplication method that original samples of minority class are appended to

datasets again and compared it with the SMOTE500, and showed that the duplication method was better than SMOTE500 [8]. Pelayo and Dick used SMOTE technique to over-sample the minority class samples, so found that SMOTE can improve the accuracy of predictors by 23% and it is not necessary to use more than SMOTE300 when using C4.5 [9]. Menzies *et al.* proposed a micro-sampling which used as low as 50 randomly selected instances (for example, 25 defective modules and 25 nondefective modules), and showed that the proposed method was effective [5].

Thirdly, some researchers tried to use ensemble learning to improve the prediction performance for class-imbalanced data. Chawla *et al.* attempted to combine SMOTE algorithm with boosting procedure to improve the accuracy of prediction [10]. Wang *et al.* proposed the AdaBoost.NC which used the error correlation information into the weights of training data and found that it was better than standard AdaBoost [15]. Wang improved the AdaBoost.NC and proposed a new version of AdaBoost by adjusting the parameters during training [4]. Laradji *et al.* attempted to use average probability ensemble (APE) learning to solve class-imbalance problem [17]. Khoshgoftaar *et al.* studied about SMOTEBoost [10] and Random Under-Sampling (RUS) Boost [16], and compared their performance, and showed that bagging was better than boosting for noisy and class-imbalanced data [18].

In a word, the research for solving the class-imbalanced problem in software defect prediction is performed at two levels: data and algorithm levels [4]. But, the algorithm depends on the data structure and the improvement of training data is more significant than the choice of algorithm [5]. From this point, we focus on the sampling technologies, especially SMOTE algorithm. If the over-sampling by duplication is used, the over-sampled data is the same as the original data. Thus, in case of 10-fold cross-validation [24], one of two same samples is used to train and the other is used to test. This is not desirable. If under-sampling is used, many of nondefective modules which may have some information will be removed and the accuracy of the prediction model will be affected [20]. Since SMOTE uses the original data to synthesize the new data (which is not the same as original data), SMOTE may be appropriate for sampling the training data. There are some problems which are faced in using the SMOTE algorithm. Over-sampling can increase not only the TP rate, but also the FP rate. Thus, over-sampling by SMOTE should be evaluated if it is effective or not. And the influence of the number of neighbors should be evaluated when using the SMOTE algorithm. However, there has not been any research about it.

In this paper, we studied about some problems that are faced in software defect prediction using SMOTE. We performed experiments for investigating how they, the percentage of appended minority class and the number of nearest neighbors, influenced the performance of software defect prediction. We used paired t-test to analyze the statistical significance of results. Also, we proposed the conceptions of effectiveness and ineffectiveness of over-sampling, and used them to define evaluation

criteria if over-sampling is effective or not. The datasets used in this paper are from the PROMISE repository [41]. We selected commonly-used classifiers to compare the performance. The results showed that the percentage of appended minority class and number of neighbors influenced the prediction performance of software defect prediction. Also, results showed that over-sampling by SMOTE was effective in case of Random Forests, Bagging, Decision Table, Ibk and J48.

The rest of this paper is organized as follows: Sec. 2 describes about sampling technologies, commonly-used classifiers, and evaluation metrics for classification in brief. In Sec. 3, we proposed the evaluation criteria for the effectiveness of over-sampling. In Sec. 4, we describe our experiment design. Section 5 analyzes the results of experiments, and Sec. 6 describes the threats to validity. In Sec. 7, we give conclusions.

2. Related Work

2.1. *Sampling strategies*

Sampling technology plays significant roles in solving the class-imbalance. Sampling technology includes over-sampling and under-sampling. Random over-sampling is a sampling technology which randomly selects the samples of minority class and adds them to the datasets [12]. In random under-sampling, randomly selects the samples of majority class and removes them from the datasets, so balance between two classes. But random over-sampling and under-sampling have some problems. If many samples of majority class are removed, we may lose the valuable information for that class. If we append many samples of minority class to the datasets, this leads to the over-fitting of minority class [20].

There are many variants of over-sampling and under-sampling. Kubat and Matwin proposed a one-sided selection under-sampling technology, which selectively removed only negative samples keeping all the positive samples [19]. Laurikkala proposed a new under-sampling technology which uses the neighborhood cleaning rule (NCL). In this method, they computed three nearest neighbors for each sample. And then if the sample is a majority class, they removed the sample, otherwise they removed the samples of majority class among the three neighbors [22]. There was an attempt to combine an under-sampling with an over-sampling for balancing imbalanced data [21]. Nevertheless, the combination of under-sampling and over-sampling could also not avoid their shortcomings [11]. Pelayo and Dick compared several sampling techniques and found that at 0.05 of significance level, the effect of under-sampling was significant and the effect of over-sampling was not significant [38].

SMOTE algorithm is a kind of over-sampling which uses the original samples of minority class to synthesize the new ones close to the given sample of minority class [10, 11]. SMOTE algorithm is more useful than random over-sampling and

under-sampling for solving class-imbalance [23]. The main procedure of SMOTE algorithm is as follows.

```
Algorithm SMOTE(T, N, k) (Source: [11])
Input: Number of minority class samples T; Amount of SMOTE N%;
Number of nearest neighbors k
Output: (N/100) * T synthetic minority class samples
(1)  if N < 100
(2)   then Randomize the T minority class samples
(3)   T = (N/100) * T
(4)   N = 100
(5)  endif
(6)  N = (int)(N/100)
(7)  k = Number of nearest neighbors
(8)  numofattrs = Number of attributes
(9)  Sample[ ][ ]: array for original minority class
(10) newindex: keeps a count of number of synthetic samples
generated, initialized to 0
(11) Synthetic[ ][ ]: array for synthetic samples
(12) for i = 1 to T
(13)  Populate(N, i, nnarray)
(14) endfor
Populate(N, i, nnarray) // Function to generate the synthetic
samples.
(15) while N ≠ 0
(16)   // Choose a random number between 1 and k, call it nn. This
step chooses one of the k nearest neighbors of i.
(17)   for attr = 1 to numattrs
(18)     dif = Sample[nnarray[nn]][attr] − Sample[i][attr]
(19)     gap = random number between 0 and 1
(20)     Synthetic[newindex][attr] = Sample[i][attr] + gap * dif
(21)   endfor
(22)   newindex++
(23)   N = N − 1
(24) endwhile
(25) return
```

Example: There are four samples: (2, 3), (5, 7), (6, 6), (9, 7). Consider a sample (2, 3) and let $k = 2$. In this case, the 2-nearest neighbors of the sample (2, 3) are, respectively, the sample (5, 7) and the sample (6, 6). Suppose that the sample (5, 7) is selected among the two neighbors.

$$x_1 = 2, \quad y_1 = 5, \quad y_1 - x_1 = 3, \quad x_2 = 3, \quad y_1 = 7, \quad y_2 - x_2 = 4$$

The new sample will be generated as follows: $(\mathbf{s}_1, \mathbf{s}_2) = (2, 3) + \mathbf{rand}(\,) \cdot (3, 4)$.

In fact, it is difficult to observe the same results as the previous one. The samples of a class will be similar to each other; therefore, a future observation will be able to be similar to the previous one. Due to this, SMOTE is able to be appropriate for over-sampling because the synthesized samples are close to the original samples of minority class. From this point, we are interested in the SMOTE algorithm. (For SMOTE algorithm, more details are described in [11].)

2.2. *Commonly-used classifiers*

Classifier is an important part of defect predictor. There are some classifiers used in software defect prediction. Commonly-used classifiers in software defect prediction are listed in Table 1 [2].

Table 1. Commonly-used classifiers in software defect prediction.

No.	Classifier	Description
1	J48	J48 is a Java implementation of C4.5. C4.5 [28] is a typical method of decision tree induction which performs the classification in such a way that constructs the decision trees from class-labeled training data.
2	Random Forests	Random Forests [29] is a kind of ensemble learning method which several decision trees is randomly learned.
3	Decision Table	Decision table [30] is one of supervised learning algorithms and use decision table with a default rule map to majority class.
4	Naïve Bayes	Naïve Bayes [31] classifier is a probabilistic classifier based on the Bayes' theorem. This classifier use the probability that a given instance belongs to a class to predict to which class belongs.
5	Logistic regression	Logistic regression [32] is a statistical method to predict the probability of categorical dependent variable using a linear combination of independent variables.
6	SMO	SMO [33, 34] is a kind of Support Vector Machines using sequential minimal optimization.
7	Bagging	Bagging is one of the ensemble methods, which combines a series of several base classifiers in order to create an improved composite classifier [35].
8	IBk	IBk is extension of nearest-neighbor classifiers, which nearest-neighbor classifiers compare a given data with training data and calculate how similar they are each other [36].

2.3. *Prediction performance metrics*

Confusion matrix is typically used in evaluating the classification model. It is shown in Fig. 1. TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives. The larger the TP and TN, the more accurate the classifier is. All the performance metrics are derived from the confusion matrix [24].

	Predicted Positive	Predicted Negative	total
Actual Positive	TP	FN	P
Actual Negative	FP	TN	N
Total	P'	N'	

Fig. 1. Confusion matrix.

In general, the accuracy of the classifier (it is also called the recognition rate) is as follows:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}. \tag{1}$$

True positive rate TP rate (it is also called the sensitivity) is referred to as the percentage of defective modules that are correctly classified by the classifier in the actually defective modules. It is defined as follows:

$$\text{TP\_rate} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{2}$$

False positive rate FP rate is the proportion of nondefective modules which is incorrectly classified as defective modules. It is defined as follows:

$$\text{FP\_rate} = \frac{\text{FP}}{\text{FP} + \text{TN}}. \tag{3}$$

The larger the TP rate and the smaller the FP rate, the more accurate the classifier is. It is a main goal for classification to increase the TP rate and decrease the FP rate, but there is a trade-off between TP rate and FP rate.

When evaluating the performance of classification over the class-imbalance data, Eq. (1) is not suitable for representing the accuracy of the classifier because this equation does not reflect how well a classifier recognizes the instances of minority class. For class-imbalanced data, G-mean is more suitable, which is defined as the geometric means of TP rate and TN rate [39].

$$G\text{-mean} = \sqrt{\text{TP rate} \times \text{TN rate}} = \sqrt{\text{TP rate} \times (1 - \text{FP rate})}. \tag{4}$$

ROC curve is also a useful metric to evaluate the classifier performance. On the ROC curve, x-axis represents the TP rate and y-axis represents the FP rate. For a binary classification, ROC curve can be used to visualize the trade-off between the true positive rate and false positive rate. The area under the ROC curve (denoted by AUC) is a measure how accurately the classifier can work. The larger AUC value is, the more accurate the classifier is [24].

3. Evaluation Criteria for the Effectiveness of the Over-Sampling

In general, the increment of performance should be proportional to the increment of resources. The main purpose of appending minority class samples is to increase *the* TP rate. Thus, the increment of TP rate should be proportional to the percentage of appended minority class samples. Although the number of minority class samples is increased n times, if the TP rate is increased a little, such an over-sampling is not useful. From this point of view, we proposed the evaluation criteria whether over-sampling is effective or not. We introduce the following concepts to define the evaluation criteria whether over-sampling is effective or not.

Definition 1. The recognition rate for minority class samples is the ratio of the number of correctly classified samples of minority class and total number of samples.

$$R = \frac{\mathrm{TP}}{N_{\mathrm{major}} + N_{\mathrm{minor}}}. \tag{5}$$

TP is the number of minority class samples which is correctly classified and N_{major} is the number of majority class samples, and N_{minor} is the number of minority class samples.

Definition 2. Effectiveness of the over-sampling is the ratio of the true positive rate for minority class before over-sampling and the one after over-sampling.

$$\mathrm{Eff} = \frac{\mathrm{TP\_rate}_{\mathrm{after}}}{\mathrm{TP\_rate}_{\mathrm{before}}}, \tag{6}$$

where $\mathrm{TP\_rate}_{\mathrm{after}}$ is the true positive rate for minority class after over-sampling and $\mathrm{TP\_rate}_{\mathrm{before}}$ is the true positive rate for minority class before over-sampling.

We are going to define a threshold value for the effectiveness of over-sampling. By what condition should the effectiveness of over-sampling be satisfied in order to apply over-sampling? Let us derivate the condition. In order to derivate the condition that the effectiveness of over-sampling should be satisfied, we use the following propositions.

Proposition 1. *If an over-sampling is effective, the number of correctly classified minority class samples after over-sampling should be equal to or larger than n times of the one before over-sampling.*

$$\mathrm{TP}'' > n \cdot \mathrm{TP}'. \tag{7}$$

Here, n is the ratio of the number of minority class samples with over-sampling and the number of minority class samples without using over-sampling.

This is reasonable. After an over-sampling, the total number of minority class samples is as n times as before. If an over-sampling is effective, the number of the correctly classified minority class samples after over-sampling will be n times more

than the number of the samples of minority class correctly classified before over-sampling. For example, there are 90 samples of majority class and 10 samples of minority class. Before over-sampling, suppose that 5 samples of minority class are correctly classified. If we use SMOTE100% to over-sample the samples of minority class, there will be 10 samples of minority class which were appended to datasets. Then among the 10 samples of minority class, more than 5 samples of minority class should be correctly classified by the classifier. The appended samples of minority class are close to original samples of minority class. Therefore, the number of minority class samples correctly classified after over-sampling should be 2 times or larger than the number of minority class samples correctly classified.

Definition 3. The recognition increment rate is the ratio of the recognition rates for minority class after and before over-sampling.

$$\delta = \frac{R_{\text{after}}}{R_{\text{before}}}, \tag{8}$$

where R_{after} is the recognition rate for minority class after over-sampling and R_{before} is the recognition rate for minority class before over-sampling.

Proposition 2. *If an over-sampling is effective, the effectiveness of the over-sampling should be larger than the recognition improvement rate* δ.

$$\text{Eff} > \delta. \tag{9}$$

This is reasonable. *TP rate* for the minority class after an over-sampling should be increased considerably. If the TP rate is not increased after the over-sampling, we cannot say that such an over-sampling is effective and cannot use such a non-effective over-sampling. That is to say, if the over-sampling is effective, TP rate for the minority class should be increased reasonably according to the percentage of the appended minority class samples. The effectiveness of over-sampling reflects by what times TP rate for minority class is improved. The recognition increment rate reflects by what times the recognition rate for minority class is improved among the total samples. As we can see from Definitions 2 and 3, the two concepts are a little similar to each other. Thus, we consider that the recognition increment rate can be used as evaluation criteria.

Let us derivate the value of δ. The derivation processes of δ are as follows.

From Definition 1, $R_{\text{after}} = \frac{\text{TP}''}{N_{\text{major}}+n\cdot N_{\text{minor}}}$ and $R_{\text{before}} = \frac{\text{TP}'}{N_{\text{major}}+N_{\text{minor}}}$. From Eq. (7) and Proposition 1, the following equation holds:

$$\delta = \frac{R_{\text{after}}}{R_{\text{before}}} = \frac{N_{\text{major}} + N_{\text{minor}}}{N_{\text{major}} + n \cdot N_{\text{minor}}} \cdot \frac{\text{TP}''}{\text{TP}'} > \frac{n \cdot (N_{\text{major}} + N_{\text{minor}})}{N_{\text{major}} + n \cdot N_{\text{minor}}}.$$

Let us introduce the rate of majority class samples and minority class samples- $\alpha = \frac{N_{\text{minor}}}{N_{\text{major}}}$. Hence, from the above two equations, the greatest lower bound of δ is as

follows:

$$\inf(\delta) = \frac{n \cdot (1+\alpha)}{1+n \cdot \alpha}. \tag{10}$$

From Eqs. (8) and (9), we can rewrite Proposition 2 as follows:

$$\text{Eff} > \frac{n \cdot (1+\alpha)}{1+n \cdot \alpha}. \tag{11}$$

Example: Suppose that there are 90 samples of majority class and 10 samples of minority class and we use SMOTE100% to over-sample the dataset. Let the true positive rate for minority class after over-sampling be 0.6 and the one before over-sampling be 0.4. In this case, let us evaluate whether this over-sampling is effective or not. Form the above conditions, $n = 2$ and $\alpha = 1/9$, and $\delta = \frac{n\cdot(1+\alpha)}{1+n\cdot\alpha} = \frac{2\cdot(1+1/9)}{2+1\cdot(1/9)} = \frac{20}{11} \approx 1.818$. Therefore, if the over-sampling is effective, the effectiveness of over-sampling should be larger than 1.818. However, the effectiveness of this over-sampling is equal to 1.5. ($\frac{0.6}{0.4} = 1.5$) Therefore, we conclude that this over-sampling is not effective because the effectiveness of this over-sampling is smaller than $\delta = 1.818$.

Proposition 2 is necessary to evaluate whether over-sampling is effective or not, but it is not enough, because over-sampling may cause the increment of FP rate. Even though the TP rate is improved by over-sampling, if the FP rate is increased highly, one cannot say that such an over-sampling is effective. Effective over-sampling should not only increase the TP rate, but also decrease the FP rate. However, FP rate may be also increased when using over-sampling. Thus, we introduce the ineffectiveness of over-sampling to evaluate if over-sampling is ineffective or not.

Definition 4. Ineffectiveness of the over-sampling is the ratio of the false positive rate for minority class before over-sampling and the one after over-sampling.

$$\text{Ineff} = \frac{\text{FP\_rate}_{\text{after}}}{\text{FP\_rate}_{\text{before}}}, \tag{12}$$

where $\text{FP\_rate}_{\text{after}}$ is the false positive rate for minority class after over-sampling and $\text{FP\_rate}_{\text{before}}$ is the false positive rate for minority class before over-sampling.

Proposition 3. *If an over-sampling is ineffective, the ineffectiveness of over-sampling should be larger than the effectiveness of over-sampling*

$$\text{Ineff} \geq \text{Eff}. \tag{13}$$

From Propositions 2 and 3, we can derivate the following proposition.

Proposition 4. *An over-sampling is effective if and only if the two following conditions are satisfied.*

(1) $\text{Eff} > \frac{n\cdot(1+\alpha)}{1+n\cdot\alpha}$
(2) $\text{Ineff} < \text{Eff}$.

We can use Proposition 4 to evaluate if an over-sampling is effective or not.

4. Experiment Design

4.1. *Software defect dataset*

The software metrics datasets used in this paper are listed in Table 2. For many of the datasets, the class-imbalance rate (i.e. percentage of defective modules in total modules) does not come up to 20%.

Table 2. Datasets from PROMISE repository.

No.	Dataset	Modules	Defective modules	Nondefective modules	$\frac{\text{Defective}}{\text{Non-defective}}$	Class-imbalance rate
1	CM1	327	42	285	0.147	12.8
2	JM1	7782	1672	6110	0.274	21.5
3	KC2	522	107	415	0.258	20.5
4	KC3	194	36	158	0.228	18.6
5	MC1	1988	46	1942	0.024	2.3
6	MC2	125	44	81	0.543	35.2
7	MW1	253	27	226	0.119	10.7
8	PC1	705	61	644	0.095	8.7
9	PC2	745	16	729	0.022	2.1
10	PC3	1077	134	943	0.142	12.4
11	PC4	1458	178	1280	0.139	12.2
12	PC5	17186	516	16670	0.031	3.0
13	AR1	121	9	112	0.080	7.4
14	AR3	63	8	55	0.145	12.7
15	AR4	107	20	87	0.230	18.7
16	AR5	36	8	28	0.286	22.2
17	AR6	101	15	86	0.174	14.9

4.2. *Experiment design*

We use WEKA 3.7.13 to do experiments over the datasets in Table 2. In this experiment, we use the 10-fold cross-validation [24] to evaluate the performance of software defect prediction. For all the classifiers, all their parameters are set by defaults in WEKA tool.

The experiment is designed as follows: First, we select a dataset among the datasets used in this study. Second, we set the percentage of appended minority class as 100%, 200%, 300%, 400% and 500% in turn and set the number of neighbors as 1–5 in turn. Third, we use SMOTE to over-sample the selected dataset. Fourth,

we select the classifiers used in this study in turn to do prediction. Fifth, we use paired t-test [40] to analyze the results. The experimental framework is shown in Fig. 2.

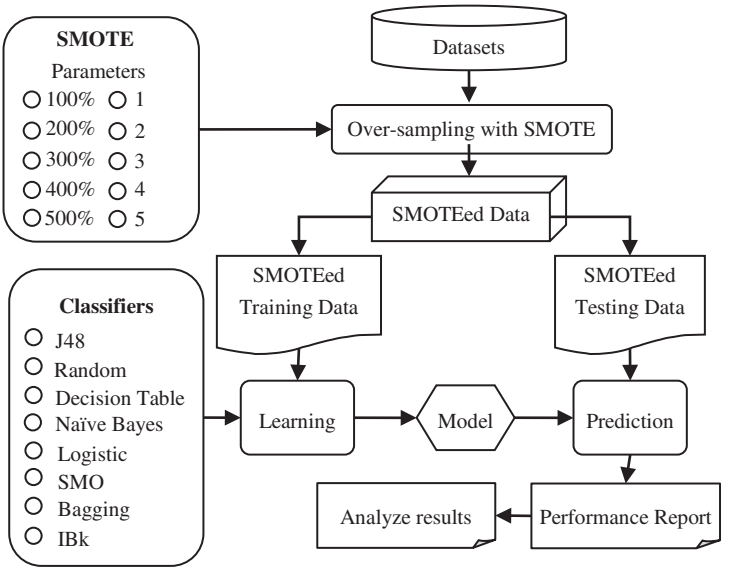

Fig. 2. Experiment framework.

4.3. *Research questions*

RQ1: How does the number of nearest neighbors influence the prediction performance of each classifier?
If the number of nearest neighbors is large, minority class samples may become distributive. This may influence the prediction performance. We verify how the number of nearest neighbors influence the prediction performance.

RQ2: How does the percentage of appended minority class influence the prediction performance of each classifier?
Over-sampling by SMOTE will make classifiers recognize minority class well, but it will not be able to improve the performance infinitely. We verify if the prediction performance can be improved until the percentage of appended minority class comes up to what percentage.

RQ3: Is the over-sampling by SMOTE effective for each classifier?
The improvement of performance should be proportional to the increment of appended minority class samples. We use our evaluation criteria to verify if over-sampling by SMOTE is effective or not for each classifier.

RQ4: Which classifier is better when using SMOTE?
For all classifiers, the improvement of performance will be different from each other. We compare the prediction performance of each classifier when using SMOTE.

5. Results and Analysis

Experiment results are shown in Tables 3–7. In Tables 3, 4 and 7, "Mean" means the average over 17 datasets, "sd" means the standard derivation, and p represents the mark which means whether the p-value of paired t-test is smaller than 0.05 or not. The mark "○" represents that p-value is smaller than 0.05, which means the

Table 3. G-mean of each classifier according to the number of nearest neighbors and the appended percentage of minority class and paired t-test.

		Mean(sd)		Mean(sd)		Mean(sd)		Mean(sd)		Mean(sd)	
	n	100%	p	200%	p	300%	p	400%	p	500%	p
Bag	1	0.692(0.136)		0.793(0.098)		0.844(0.046)		0.855(0.069)		0.890(0.033)	
	2	0.688(0.144)	×	0.783(0.085)	×	0.827(0.057)	○	0.845(0.098)	×	0.870(0.045)	○
	3	0.668(0.168)	×	0.790(0.083)	×	0.841(0.050)	×	0.845(0.074)	○	0.864(0.046)	○
	4	0.683(0.159)	×	0.774(0.085)	×	0.824(0.069)	○	0.838(0.081)	○	0.865(0.047)	○
	5	0.663(0.169)	×	0.779(0.113)	×	0.822(0.066)	○	0.841(0.072)	○	0.867(0.049)	○
DT	1	0.652(0.137)		0.752(0.083)		0.800(0.050)		0.804(0.100)		0.843(0.042)	
	2	0.647(0.152)	×	0.740(0.084)	×	0.780(0.072)	○	0.792(0.104)	○	0.825(0.068)	×
	3	0.640(0.141)	×	0.727(0.104)	○	0.781(0.064)	○	0.785(0.117)	○	0.817(0.063)	○
	4	0.655(0.128)	×	0.729(0.101)	×	0.777(0.064)	○	0.788(0.107)	○	0.816(0.083)	○
	5	0.607(0.171)	×	0.723(0.102)	○	0.781(0.062)	○	0.785(0.137)	○	0.816(0.082)	○
IBk	1	0.795(0.058)		0.843(0.037)		0.875(0.040)		0.869(0.055)		0.884(0.044)	
	2	0.779(0.050)	○	0.832(0.044)	○	0.858(0.039)	○	0.860(0.052)	○	0.874(0.046)	○
	3	0.759(0.058)	○	0.820(0.039)	○	0.848(0.038)	○	0.845(0.059)	○	0.868(0.044)	○
	4	0.767(0.061)	○	0.798(0.083)	○	0.835(0.044)	○	0.839(0.060)	○	0.859(0.041)	○
	5	0.750(0.059)	○	0.801(0.057)	○	0.835(0.038)	○	0.834(0.071)	○	0.853(0.044)	○
J48	1	0.736(0.086)		0.824(0.050)		0.841(0.055)		0.855(0.085)		0.876(0.037)	
	2	0.729(0.092)	×	0.801(0.054)	○	0.835(0.048)	×	0.835(0.073)	○	0.869(0.035)	×
	3	0.734(0.089)	×	0.789(0.056)	○	0.845(0.045)	×	0.841(0.070)	×	0.866(0.033)	×
	4	0.721(0.104)	×	0.787(0.062)	○	0.827(0.048)	×	0.831(0.077)	○	0.865(0.037)	×
	5	0.715(0.097)	×	0.797(0.059)	○	0.837(0.042)	×	0.830(0.086)	○	0.856(0.038)	○
Log	1	0.633(0.108)		0.679(0.109)		0.720(0.093)		0.720(0.127)		0.753(0.094)	
	2	0.629(0.119)	×	0.690(0.089)	×	0.720(0.095)	×	0.712(0.126)	×	0.750(0.095)	×
	3	0.635(0.100)	×	0.687(0.090)	×	0.716(0.093)	×	0.719(0.124)	×	0.742(0.102)	○
	4	0.642(0.099)	×	0.680(0.100)	×	0.706(0.100)	○	0.711(0.126)	×	0.735(0.092)	○
	5	0.634(0.092)	×	0.680(0.108)	×	0.711(0.089)	×	0.712(0.114)	×	0.739(0.098)	○
NB	1	0.612(0.122)		0.623(0.117)		0.635(0.113)		0.619(0.129)		0.636(0.117)	
	2	0.611(0.126)	×	0.626(0.117)	×	0.632(0.114)	×	0.622(0.135)	×	0.637(0.114)	×
	3	0.611(0.115)	×	0.621(0.108)	×	0.634(0.111)	×	0.619(0.131)	×	0.633(0.112)	×
	4	0.610(0.115)	×	0.626(0.113)	×	0.633(0.114)	×	0.619(0.131)	×	0.631(0.105)	×
	5	0.615(0.134)	×	0.632(0.122)	×	0.642(0.119)	×	0.629(0.133)	×	0.637(0.118)	×
RF	1	0.764(0.093)		0.857(0.049)		0.891(0.034)		0.906(0.036)		0.919(0.029)	
	2	0.752(0.111)	×	0.845(0.055)	○	0.884(0.037)	×	0.895(0.047)	○	0.912(0.031)	○
	3	0.742(0.107)	×	0.855(0.049)	×	0.880(0.043)	×	0.884(0.061)	○	0.909(0.031)	○
	4	0.736(0.118)	○	0.829(0.070)	○	0.868(0.053)	○	0.888(0.048)	○	0.903(0.034)	○
	5	0.717(0.149)	○	0.830(0.073)	○	0.869(0.052)	○	0.883(0.060)	○	0.899(0.037)	○
SMO	1	0.455(0.269)		0.590(0.256)		0.641(0.258)		0.663(0.258)		0.651(0.292)	
	2	0.444(0.264)	×	0.572(0.259)	○	0.627(0.256)	○	0.646(0.256)	○	0.643(0.297)	×
	3	0.437(0.263)	○	0.561(0.265)	×	0.621(0.256)	○	0.641(0.266)	×	0.641(0.294)	×
	4	0.436(0.259)	×	0.575(0.256)	×	0.622(0.255)	○	0.642(0.266)	○	0.633(0.289)	○
	5	0.441(0.284)	×	0.548(0.248)	○	0.629(0.252)	○	0.645(0.266)	×	0.641(0.295)	×

Table 4. *G*-mean of each classifier according to the appended percentage of minority class and paired *t*-test.

	Bag		DT		IBk		J48		Log		NB		RF		SMO	
%	Mean(sd)	p	Mean(sd)	p	Mean(sd)	p	Mean(sd)	p	Mean(sd)	p	Mean(sd)	p	Mean(sd)	p	Mean(sd)	p
0	0.349(0.235)		0.364(0.262)		0.535(0.172)		0.504(0.194)		0.470(0.180)		0.593(0.122)		0.464(0.240)		0.205(0.245)	
100	0.692(0.136)	∘	0.652(0.137)	∘	0.795(0.058)	∘	0.736(0.086)	∘	0.633(0.108)	∘	0.612(0.122)	∘	0.764(0.093)	∘	0.455(0.269)	∘
200	0.793(0.098)	∘	0.752(0.083)	∘	0.843(0.037)	∘	0.824(0.050)	∘	0.679(0.109)	∘	0.623(0.117)	×	0.857(0.049)	∘	0.590(0.256)	∘
300	0.844(0.046)	∘	0.800(0.050)	∘	0.875(0.040)	∘	0.841(0.055)	∘	0.720(0.093)	∘	0.635(0.113)	∘	0.891(0.034)	∘	0.641(0.258)	∘
400	0.855(0.069)	×	0.804(0.100)	×	0.869(0.055)	×	0.855(0.085)	×	0.720(0.127)	×	0.619(0.129)	×	0.906(0.036)	∘	0.663(0.258)	×
500	0.890(0.033)	×	0.843(0.042)	×	0.884(0.044)	×	0.876(0.037)	×	0.753(0.094)	×	0.636(0.117)	×	0.919(0.029)	∘	0.651(0.292)	×

Table 5. Evaluation criteria for effectiveness of over-sampling.

No.	Percentage	n	α	$\delta(\gamma)$
1	100	2	0.173	1.743
2	200	3	0.173	2.317
3	300	4	0.173	2.773

Table 6. Effectiveness and Ineffectiveness of over-sampling depending on the percentage.

No.	Classifier	Effectiveness			Ineffectiveness			Eff & Ineff		
SMOTE%		100	200	300	100	200	300	100	200	300
1	Bag	2.933	3.838	4.337	1.943	2.483	2.804	1	1	1
2	DT	2.327	3.062	3.472	2.068	2.706	3.068	1	1	1
3	IBk	1.898	2.165	2.330	0.842	0.942	0.951	1	0	0
4	J48	1.934	2.412	2.525	1.453	1.525	1.607	1	1	0
5	Log	1.599	1.908	2.168	1.010	1.354	1.573	0	0	0
6	NB	1.067	1.109	1.148	1.039	1.095	1.128	0	0	0
7	RF	2.243	2.813	3.069	1.627	1.887	2.173	1	1	1
8	SMO	2.896	4.478	5.398	2.823	5.413	7.427	1	0	0

Table 7. AUC values depending on the appended percentage and paired t-test.

	No SMOTE			SMOTE100			SMOTE200		
Rank	Classifier	Mean(sd)	p	Classifier	Mean(sd)	p	Classifier	Mean(sd)	p
1	RF	0.802(0.086)		RF	0.924(0.041)		RF	0.949(0.026)	
2	Bag	0.770(0.103)	∘	Bag	0.870(0.056)	∘	Bag	0.905(0.044)	∘
3	NB	0.753(0.090)	×	IBk	0.819(0.067)	∘	IBk	0.844(0.056)	∘
4	Log	0.724(0.112)	×	DT	0.807(0.077)	×	DT	0.839(0.061)	×
5	IBk	0.657(0.103)	∘	NB	0.793(0.078)	×	J48	0.828(0.054)	×
6	DT	0.654(0.159)	×	Log	0.780(0.090)	×	Log	0.807(0.078)	×
7	J48	0.632(0.106)	×	J48	0.780(0.075)	×	NB	0.794(0.081)	×
8	SMO	0.544(0.070)	∘	SMO	0.634(0.111)	∘	SMO	0.678(0.115)	∘
	SMOTE300			SMOTE400			SMOTE500		
Rank	Classifier	Mean(sd)	p	Classifier	Mean(sd)	p	Classifier	Mean(sd)	p
1	RF	0.961(0.022)		RF	0.969(0.018)		RF	0.975(0.014)	
2	Bag	0.930(0.030)	∘	Bag	0.938(0.025)	∘	Bag	0.945(0.025)	∘
3	DT	0.870(0.054)	∘	DT	0.888(0.045)	∘	DT	0.900(0.048)	∘
4	IBk	0.864(0.053)	×	IBk	0.876(0.053)	×	J48	0.883(0.032)	×
5	J48	0.860(0.044)	×	J48	0.868(0.040)	×	IBk	0.882(0.047)	×
6	Log	0.834(0.065)	×	Log	0.834(0.066)	×	Log	0.849(0.066)	∘
7	NB	0.802(0.081)	∘	NB	0.805(0.080)	∘	NB	0.805(0.078)	∘
8	SMO	0.709(0.109)	∘	SMO	0.722(0.117)	∘	SMO	0.735(0.125)	∘

hypothesis is received. The mark "×" represents that p-value is not smaller than 0.05, which means the hypothesis is rejected. In Table 7, "Mean" means the average of AUC (area under c) over 17 datasets. We used the paired t-test to test the statistical significance of results at the significance level of 0.05.

5.1. *Evaluation depending on the number of nearest neighbors used in SMOTE*

As you can see from Table 3, when the number of nearest neighbors was equal to 1, in most of cases, the prediction performance (G-mean) was better than the others. However, the difference of performance was small. Thus, we hypothesized that when the number of nearest neighbors is equal to 1, G-mean is the best, and then we tested the statistical significance of this hypothesis. As a result, when using IBk, in all cases, the hypothesis was received, and when using Naïve Bayes, in all cases, the hypothesis was rejected. This means IBk was influenced by the number of nearest neighbors in case of using SMOTE and Naïve Bayes was never influenced. For Bagging, Decision Table and Random Forests, when using SMOTE300, 400 and 500, there were tendencies that in many cases, the hypothesis was received. For J48 and SMO, there were no clear tendencies. Logistic was also not influenced by the number of nearest neighbors in most of cases. Therefore, in software defect prediction using SMOTE, the number of nearest neighbors should be considered according to classifiers and percentages of appended minority class.

5.2. *Evaluation depending on the appended percentage of minority class samples*

We used the G-mean when the number of nearest neighbors is equal to 1. We hypothesized that *G-mean* was increased with the increment of the appended percentage of minority class, and then tested this hypothesis. As a result, for Bagging, Decision Tree, IBk, J48, Logistic and SMO, there were statistical differences between No SMOTE and SMOTE100, SMOTE100 and SMOTE200, and SMOTE200 and SMOTE300. There were no statistical differences between SMOTE300 and SMOTE400, and SMOTE400 and SMOTE500. This means that it is not necessary to append more than 300% of minority class samples for these classifiers. For Naïve Bayes, there was a statistical difference between No SMOTE and SMOTE100, but there was no difference between SMOTE100 and SMOTE200. This means that it is not necessary to append the minority class samples of more than 100%.

5.3. *Evaluation depending on our evaluation criteria*

The critical values for evaluation criteria are shown in Table 5. In Table 5, α is the average of ratios between minority class and majority class over the 17 datasets. Since there were no statistical differences in using more than SMOTE300, we evaluated until SMOTE300. Table 6 shows the effectiveness and ineffectiveness of oversampling by SMOTE. In Table 6, "Eff & Ineff" means that the logical conjunction of two conditions in Proposition 4. From Table 6, we could see that Bagging, Decision Table and Random Forests were effective. IBk and SMO were effective only when using SMOTE100 and J48 was effective only when using SMOTE100 and SMOTE200. Logistic and Naïve Bayes were not effective.

5.4. *Comparison of classifiers*

We used the AUC values for comparing the performance of classifiers. We ranked the classifiers according to AUC values and tested the statistical significance of the rank. As a result, in most of the cases, Random Forests and Bagging were better than the others, and SMO was worst. There were no clear statistical differences between the other classifiers.

6. Threats to Validity

This study includes the threats to validity. In this section, we will discuss the possible threats to validity.

In this study, Propositions 1–4 were based on empirical rule that the increment of performance should be proportional to increment of resources, and therefore they can pose threats to internal validity. When using the over-sampling, it is not reasonable that the samples of minority class are infinitely appended to a dataset without any constraint. This is because that if the number of the appended samples wins the number of the original samples, the dataset may lose the original nature of itself. Thus, it is necessary to define the evaluation criteria. The effectiveness of over-sampling and the recognition increment rate are similar to each other in the means that represents by what times the recognition rate for minority class is improved after over-sampling. The true positive rate is obtained by a classifier, but the greatest lower bound of the recognition increment rate is obtained if the dataset is given. From this point of view, we selected the recognition increment rate as the evaluation criteria. Similarly, we defined the ineffectiveness of over-sampling, and used these conceptions to define evaluation criteria whether the over-sampling is ineffective or not.

Threats to external validity may arise if the results observed for software defect prediction in using SMOTE algorithm are different for other classification problem. SMOTE algorithm is used in not only software defect prediction but also several classification problems. Since an application of SMOTE algorithm may depend on the nature of the datasets and classifiers used in the classification, the number of nearest neighbors used in SMOTE and the appended percentage of minority class should be considered according to the characteristics of the given problem and the used classifier.

7. Conclusion

In this paper, we studied about the software defect prediction using SMOTE. The main contributions of this work include the following:

(1) We examined the influence of the number of neighbors and the appended percentage of minority class, and we used paired t-test to test the statistical significance of the influence.

(2) We proposed the evaluation criteria for evaluating whether over-sampling is effective or not.
(3) We compared the prediction performance of classifiers when using SMOTE and found that Random Forests and Bagging are better than the others.

SMOTE can improve the prediction performance, but SMOTE alone cannot completely solve the class-imbalance problem in software defect prediction. In future, we are going to study about the data preprocessing method after over-sampling.

Acknowledgments

This work was supported by "the 13th Five-Year" National Science and Technology Major Project of China (Grant No. 2017YFC0702204), National Natural Science Foundation of China (Grant No. 61672191) and Harbin Science and Technology Innovation Talents Research Project (Grant No. 2016RAQXJ013). We would like to express our sincere and heartfelt thanks to the editors and reviewers.

References

1. M. J. Siers and Md. Z. Islam, Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem, *Inf. Syst.* **51** (2015) 62–71.
2. R. S. Wahono, A systematic literature review of software defect prediction: Research trends, datasets, methods and frameworks, *J. Softw. Eng.* **1**(1) (2015) 1–16.
3. Ö. F. Arara and K. Ayanba, Software defect prediction using cost-sensitive neural network, *Appl. Soft Comput.* **33** (2015) 263–277.
4. S. Wang and X. Yao, Using class imbalance learning for software defect prediction, *IEEE Trans. Reliabil.* **62**(2) (2013) 434–443.
5. T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic and Y. Jiang, Implications of ceiling effects in defect predictors, in *Int. Workshop on Predictor Models in Software Engineering*, 2008, pp. 47–54.
6. Z.-H. Zhou and X.-Y. Liu, Training cost-sensitive neural networks with methods addressing the class imbalance problem, *IEEE Trans. Knowl. Data Eng.* **18**(1) (2006) 63–77.
7. J. Zheng, Cost-sensitive boosting neural networks for software defect prediction, *Expert Syst. Appl.* **37**(6) (2010) 4537–4543.
8. R. Shatnawi, Improving software fault-prediction for imbalanced data, in *Int. Conf. Innovations in Information Technology*, 2012, pp. 54–59.
9. L. Pelayo and S. Dick, Applying novel resampling strategies to software defect prediction, in *Annual Meeting of the North American Fuzzy Information Processing Society*, 2007, pp. 69–72.
10. N. V. Chawla, A. Lazarevic, L. O. Hall and K. W. Bowyer, SMOTEBoost: Improving prediction of the minority class in boosting, *Knowledge Discovery in Databases: PKDD 2003*, LNAI Vol. 2838, 2003, pp. 107–119.
11. N. V. Chawla, K. W. Bowyer, L. O. Hell and W. P. Kegelmeyer, SMOTE: Synthetic minority over-sampling technique, *J. Artifi. Intell. Res.* **16** (2002) 321–357.

12. H. He and E. A. Garcia, Learning from imbalanced data, *IEEE Trans. Knowl. Data Eng.* **21**(9) (2009) 1263–1284.
13. D. Rodriguez, I. Harraiz and R. Harrison, Preliminary comparison of techniques for dealing with imbalance in software defect prediction, in *Int. Conf. Evalua. Assessment Softw. Eng.* **43** (2014) 1–10.
14. R. Moser, W. Pedrycz and G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in *Int. Conf. Software Engineering*, 2008, pp. 181–190.
15. S. Wang, H. Chen and X. Yao, Negative correlation learning for classification ensembles, in *Proc. Int. Joint Conf. Neural Netw., WCCI*, 2010, pp. 2893–2900.
16. C. Seiffert, T. M. Khoshgoftaar, J. V. Hulse and A. Napolitano, RUSBoost: A hybrid approach to alleviating class imbalance, *IEEE Trans. Syst. Man, Cybern. Part A, Syst. Humans* **40**(1) (2010) 185–197.
17. I. H. Laradji, M. Alshayeb and L. Ghout, Software defect prediction using ensemble learning on selected features, *Inf. Softw. Technol.* **58** (2015) 388–402.
18. T. M. Khoshgoftaar, J. V. Hulse and A. Napolitano, Comparing boosting and bagging techniques with noisy and imbalanced data, *IEEE Trans. Syst. Man, and Cybern. Part A, Syst. Humans* **41**(3) (2011) 552–568.
19. M. Kubat and S. Matwin, Addressing the curse of imbalanced training sets: One sided selection, in *Proc. Fourteenth Int. Conf. Machine Learning*, 1997, pp. 179–186.
20. N. V. Chawla, Data mining for imbalanced datasets: An overview, *Data Mining and Knowledge Discovery Handbook* (Springer Science & Business Media, 2010), pp. 875–886.
21. A. Estabrooks, T. Jo and N. Japkowicz, A multiple resampling method for learning from imbalanced data sets, *Comput. Intell.* **20**(1) (2004) 18–36.
22. J. Laurikkala, Improving identification of difficult small classes by balancing class distribution, *AIME2001, LNAI2101*, 2001, pp. 63–66.
23. G. E. A. P. A. Batista, R. C. Prati and M. C. Monard, A study of the behavior of several methods for balancing machine learning training data, *SIGKDD Explor.* **6**(1) (2004) 20–29.
24. J. Han, M. Kamber and J. Pei, *Data Mining Concepts and Techniques*, 3rd edn. (Morgan Kaufmann, 2012).
25. T. J. McCabe, A complexity measure, *IEEE Trans. Softw. Eng.* **2**(4) (1976) 308–320.
26. M. H. Halstead, *Elements of Software Science* (Elsevier Science, 1977).
27. S. R. Chidamber and C. F. Kemerer, A metrics suite for object oriented design, *IEEE Trans. Softw. Eng.* **20**(6) (1994) 476–493.
28. J. Ross, *Quinlan.C4.5: Programs for Machine Learning* (Morgan Kaufmann, 1993).
29. L. Breiman, Random forests, *Mach. Learn.* **45** (2001) 5–32.
30. R. Kohavi, The power of decision tables, in *8th European Conf. Machine Learning*, 1995, pp. 174–189.
31. G. H. John and P. Langley, Estimating continuous distributions in Bayesian classifiers, *Proc. Eleventh Conference on Uncertainty in Artificial Intelligence*, 1995, pp. 338–345.
32. S. Le Cessie and J. C. Van Houwelingen, Ridge estimators in logistic regression, *Appl. Stat.* **41**(1) (1992) 191–201.
33. S. S. Keerthi, S. K. Shevade, C. Bhattacharyya and K. R. K. Murthy, Improvements to Platt's SMO algorithm for SVM classifier design, *Neural Comput.* **13**(3) (2001) 637–649.
34. J. Platt, Fast training of support vector machines using sequential minimal optimizaion, *Advances in Kernel Methods* (Philomel Books, 1999), pp. 185–208.
35. L. Breiman, Bagging predictors, *Mach. Learn.* **24**(2) (1996) 123–140.
36. D. Aha and D. Kibler, Instance-based learning algorithms, *Mach. Learn.* **6** (1991) 37–66.

37. A. Okutan and O. T. Yıldız, Software defect prediction using Bayesian networks, *Empiri. Softw. Eng.* **19** (2014) 154–181.
38. L. Pelayo and S. Dick, Evaluating stratification alternatives to improve software defect prediction, *IEEE Trans. Reliabil.* **61** (2012) 516–525.
39. M. Kubat and S. Matwin, Addressing the curse of imbalanced training sets: One-sided selection, in *Int. Conf. Machine Learning*, 1997, pp. 179–186.
40. J. Demšar, Statistical comparisons of classifiers over multiple data sets, *J. Mach. Learn. Res.* **7** (2006) 1–30.
41. http://openscience.us/repo/defect/mccabehalseted/.