Empir Software Eng
https://doi.org/10.1007/s10664-018-9633-6

On the relative value of data resampling approaches for software defect prediction

Kwabena Ebo Bennin[1] · Jacky W. Keung[1] · Akito Monden[2]

Abstract Software defect data sets are typically characterized by an unbalanced class distribution where the defective modules are fewer than the non-defective modules. Prediction performances of defect prediction models are detrimentally affected by the skewed distribution of the faulty minority modules in the data set since most algorithms assume both classes in the data set to be equally balanced. Resampling approaches address this concern by modifying the class distribution to balance the minority and majority class distribution. However, very little is known about the best distribution for attaining high performance especially in a more practical scenario. There are still inconclusive results pertaining to the suitable ratio of defect and clean instances (*Pfp*), the statistical and practical impacts of resampling approaches on prediction performance and the more stable resampling approach across several performance measures. To assess the impact of resampling approaches, we investigated the bias and effect of commonly used resampling approaches on prediction accuracy in software defect prediction. Analyzes of six resampling approaches on 40 releases of 20 open-source projects across five performance measures and five imbalance rates were performed. The experimental results obtained indicate that there were statistical differences between the prediction results with and without resampling methods when evaluated with the

Communicated by: Nachiappan Nagappan

✉ Kwabena Ebo Bennin
kebennin2-c@my.cityu.edu.hk

Jacky W. Keung
Jacky.Keung@cityu.edu.hk

Akito Monden
monden@okayama-u.ac.jp

[1] Department of Computer Science, City University of Hong Kong, Kowloon Tong, Hong Kong

[2] Graduate School of Natural Science and Technology, Okayama University, Okayama, Japan

geometric-mean, recall(*pd*), probability of false alarms(*pf*) and *balance* performance measures. However, resampling methods could not improve the AUC values across all prediction models implying that resampling methods can help in defect classification but not defect prioritization. A stable *Pfp* rate was dependent on the performance measure used. Lower *Pfp* rates are required for lower *pf* values while higher *Pfp* values are required for higher *pd* values. Random Under-Sampling and Borderline-SMOTE proved to be the more stable resampling method across several performance measures among the studied resampling methods. Performance of resampling methods are dependent on the imbalance ratio, evaluation measure and to some extent the prediction model. Newer oversampling methods should aim at generating relevant and informative data samples and not just increasing the minority samples.

Keywords Software defect prediction · Imbalanced data · Data resampling approaches · Class imbalance · Empirical study

1 Introduction

Software defect prediction models can conveniently identify software modules or classes that are likely to be defective. This helps in focusing on the specific modules, which really needs attention with regards to code inspection and efficiently prioritize and allocate the limited testing resources to those modules identified as likely to be defective (Wang and Yao 2013; Menzies et al. 2007).

Defect prediction datasets are usually skewed or imbalanced with the non fault-prone modules dominating the fault-prone modules (Wang and Yao 2013). This is not so surprising as several high quality assurance activities are employed during the testing phase of a software project hence drastically reducing the total number of faults in the software (Drown et al. 2009). A defect prediction model trained on such a skewed data set will thus be more biased toward the non fault-prone modules or majority class and inadvertently disregard the minority or fault-prone modules (Weiss and Provost 2001). This is referred to as class imbalance learning in machine learning and data mining domain and using such a trained model to classify the target project will most often predict a fault-prone module as non fault-prone. Researchers and software quality teams are however more interested in the minority faulty classes (Seiffert et al. 2010).

In the past decade, different approaches proposed to tackle the issue of class imbalanced data set by researchers include: data resampling techniques, assigning specific cost to the classes aimed at reducing the classification cost (Pazzani et al. 1994; Sun et al. 2007; Domingos 1999) and use of ensemble learning techniques (Laradji et al. 2015; Wong et al. 2013). However, adoption of data resampling techniques as a measure to alleviate the issue of imbalanced datasets by adding new instances to the minority class or removing majority instances known as over and under-sampling respectively is very common due to the ease of implementation and relatively positive effect on classification performance. Previous studies (García et al. 2012; Shanab et al. 2012; Japkowicz and Stephen 2002) conclude that oversampling approaches perform better than undersampling approaches due to the fact that no information is lost and as such, most of the recent resampling approaches proposed in the literature are all based on oversampling (He et al. 2008; Han et al. 2005; Chawla et al. 2002; Bennin et al. 2017). Figure 1 shows the impact of oversampling on a classification model. The shift in decision boundary built before and after applying oversampling

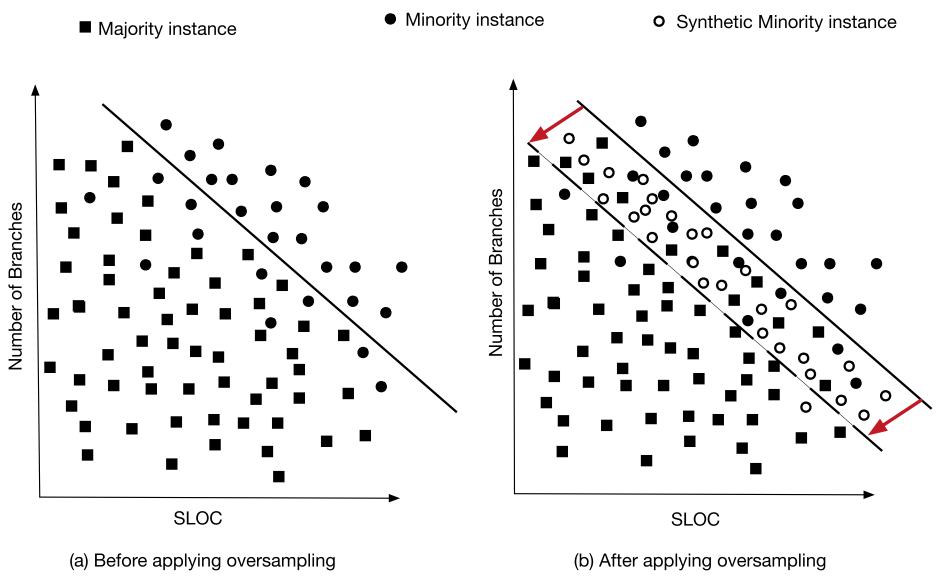

Fig. 1 Impact of Sampling on a classification model

is clear, implying that oversampling has a significant effect on the classification performance. Specifically, before oversampling the data, the decision boundary of the classifier is largely induced by the dominating non-defective modules. This model will over predict the non-defective modules and will have a low probability of predicting the defective modules. However, after oversampling the data, the classification model built will now have an equal probability of predicting both the defective and non-defective modules. Consequently, the performance of the model is largely improved.

Despite the obvious fact that class imbalance is one of the critical factors that affects the performance of software defect prediction performance (Hall et al. 2012; Arisholm et al. 2010; Sun et al. 2012), most studies (Wang and Yao 2013; Riquelme et al. 2008; Pelayo and Dick 2007) focus on applying or proposing data resampling approaches without considering the amount of resampling or percentage of fault-prone modules (*Pfp*) actually required. The objective functions of most machine learning algorithms assume that datasets are balanced prior to training (Chawla et al. 2002; Phung et al. 2009; Laradji et al. 2015). The question we therefore ask ourselves is “Is balanced the best distribution?” Agrawal and Menzies (Agrawal and Menzies 2017) noted that the performance of prediction models can be significantly improved when the parameters of SMOTE sampling method are configured. Their study suggests that configuring the parameters of sampling methods can improve the performance of prediction models. Different sampling methods however, have different configurable parameters and it is impractical to consider and configure all these parameters during model construction. Nevertheless, the *Pfp* which controls the amount of data added or deleted from the minority class samples respectively is a constant parameter present in all sampling methods. This study focuses on the effect of the *Pfp* on the performance of prediction models considering other sampling methods including the SMOTE method. Additionally, empirical research conducted till date used different evaluation measures and did not statistically evaluate the results leading to the inconclusive stability as to the best resampling approach and *Pfp*.

We thus perform a systematic analysis to examine the extent data resampling approaches improve prediction performance and the rate of resampling required. Specifically, we conduct experiments aimed at answering the following research questions:

RQ1: Is the effect of resampling approaches on prediction models statistically and practically significant regarding the evaluation metric and dataset used?

RQ2: To what degree (percentage of fault-prone modules *Pfp*) should one resample or balance the original data set?

RQ3: Which are the high-performing resampling approaches for defect prediction? Based on the above research questions, extensive empirical experiments were conducted on 40 releases of 20 software projects. 20 of these projects are available at the PROMISE repository while the other 20 open source projects were manually extracted. We considered 6 data resampling approaches: Synthetic Minority Over Sampling Technique (SMOTE)(Chawla et al. 2002), Borderline-SMOTE (Han et al. 2005), Safe-level SMOTE (Bunkhumpornpat et al. 2009), ADASYN (He et al. 2008), Random Over Sampling (ROS), Random Under Sampling (RUS), 5 defect prediction models (KNN, SVM, C4.5, RF and NNET) and 5 evaluation performance measures (AUC, *g-mean*, *balance*, *pd* and *pf*). The results are analyzed by combining the data resampling approaches with the prediction models. As an extension of our previous work (Bennin et al. 2017), this study is the first to empirically validate several resampling methods across more data sets and statistically test the significance using robust statistical measures. Two more research questions have been included and investigated in this study. The study examines different *Pfp* values with the aim of finding the most stable value for data resampling methods considering more datasets, resampling approaches and performance measures. Additionally, we contribute by providing recommendations and guidelines on which data resampling method to use or avoid for software quality teams and researchers. In summary, this work provides the following contributions.

1. We present the most stable and effective data resampling method for SDP-Software Defect Prediction.
2. Empirically evaluate the practical effects of different resampling rates (*Pfp*) on defect prediction performance.
3. The first paper to examine a large collection of resampling techniques and find a stable *Pfp* rate for data resampling methods.

We encourage the use of resampling methods only for defective modules classification and not prioritization. To help improve defect prioritization, resampling methods which could concurrently achieve low *pf* and high *pd* is needed. Overall, the simple RUS method outperformed the complex SMOTE and it's variants. We thus recommend the use of RUS when large data samples of a particular project exist. SMOTE and other variants of SMOTE does improve performance but also does increase the false alarm rate (*pf*).

The organization of this paper is as follows. Section 2 reviews the most closely related work in connection with imbalanced data set and resampling techniques. Section 3 discusses the background of the resampling methods used. In Section 4, we describe the experimental methodology and performance evaluation measures used in validating our approach. The experimental results are presented in Section 5 and Section 6 discusses the impacts, some limitations of our study. Finally, a summary of the paper is presented in Section 8.

2 Related Work

Class imbalance and its effect on the performance of software defect prediction models has received little attention until recently. Approaches proposed to handle the class imbalance issue could be categorized into three. They are: (a) Resampling the original datasets (data resampling approaches) by rebalancing the distribution of the class (Chawla et al. 2002; Kubat et al. 1997), (b) Assigning specific cost to the classes to reduce classification cost (Sun et al. 2007; Liu et al. 2014; Zheng 2010) and (c) applying ensemble learning techniques (Laradji et al. 2015; Wong et al. 2013). Data resampling approaches are however more prevalent in literature because it is not dependent on the prediction model used (Kamei et al. 2007). In this section, summary discussions on the application of data resampling approaches in software defect prediction studies is presented.

Kamei et al. (2007), were the first to investigate the effects of resampling methods on performance of defect prediction models. They empirically evaluated four resampling methods (random over sampling, synthetic minority over sampling, random under sampling and one-sided selection) assessed with four defect prediction models-Linear Discriminant Analysis (LDA), Logistic Regression Analysis(LRA), Neural Network (NNET) and Classification Tree (CT) on two sets of an industry legacy software. Their results showed that prediction performance of LDA and LRA were improved when the four resampling methods were applied to the data prior to model training. However, the performance of two other models - NNET and CT were not improved. The authors also observed an inconclusive result regarding the best resampling approach and the rate of resampling required to use for defect prediction. A major limitation of this work was the use of only a single dataset that is not publicly available.

Pelayo and Dick (2007) applied the SMOTE method to four NASA datasets extracted from the PROMISE repository and assessed the performance with the C4.5 decision tree model. Using the geometric mean performance measure, they observed an improvement of at least 23% on all four datasets. Similarly, Riquelme et al. (2008) also evaluated the performance of SMOTE and Random oversampling using the Naive Bayes and C4.5 models on five datasets extracted from PROMISE repository. Their results showed that the AUC performance measure is improved when the SMOTE method is applied to the datasets before training. Contrary to previous studies, a recent study by Shatnawi (2017) showed that undersampling and SMOTE do not improve the AUC performance of prediction models. Menzies et al. (2008) observed that resampling methods increase the information content of a training data which consequently improves the performance of prediction models. Although these studies indicate that resampling of defect datasets is important for better performance of prediction models, they fail to indicate the amount of resampling required. Secondly, they all validated their experiments on ten or less datasets and two or less prediction models. Several prediction models and datasets exist. Lastly, they mostly adopted the SMOTE method while there exist several other resampling methods in literature.

Other comparative studies incorporated ensemble and boosting techniques. Sun et al. (2012) proposed an ensemble method based on coding schemes that transforms the binary-class into a multi-class dataset before training a model. Comparing their method to other conventional resampling methods on 14 NASA datasets and four prediction models, the performance of C4.5, Ripper and Random Forest were improved with their ensemble method. Wang and Yao (2013) conducted an in-depth investigation of how class imbalance learning methods could improve software defect prediction performance. By comparing class imbalance learning methods of three main types (undersampling, threshold-moving and boosting-based ensembles) with two popular base classifiers (Naive Bayes and Random

Forest) on ten PROMISE datasets, their results showed that AdaBoost.NC, an ensemble method had the best performance in terms of the G-mean and AUC measures.

Contrary to these studies, our recent study (Bennin et al. 2016) on ten manually extracted open source projects and ten effort-aware prediction models showed that no resampling was better than applying Random Under Sampling and SMOTE to a dataset when an effort-aware performance measure is used. Also, Random Over Sampling outperformed SMOTE. This implies the dataset and evaluation criteria used are deciding factors in determining the best resampling method and *Pfp*. Agrawal and Menzies (2017) in their recent study conclude that for better and improved prediction performance, the parameters of SMOTE should be adjusted based on the distribution of the training data. This study focuses on using the resampling methods with their default parameters. We only alter the distribution rates (*Pfp*) of the training datasets for all sampling methods and not only SMOTE. A detailed assessment of how resampling methods improve performance and a solid resolution as to the best resampling approach and *Pfp* required is thus necessary. We assess in an extensive, systematic and comprehensive experiment, the choice of resampling methods, prediction model, evaluation criteria and its effect on the prediction performance.

3 Background

We introduce the six data resampling approaches adopted and used in this study.

3.1 SMOTE

Chawla et al. (2002) proposed the commonly known Synthetic Minority Over-sampling Technique (SMOTE) that creates synthetic data instances in the minority class region so as to shift the bias of the classifier toward the minority class. SMOTE automatically deletes some majority instances and increases the minority instances. The SMOTE algorithm generates synthetic instances by:

1. Selecting an instance from the minority class samples, for example, *x1* in Fig. 2.
2. Find the defined *k* minority class nearest neighbors instances of *x1*. (k=5)
3. New synthetic instances are created along the line segment that join each neighbor instance and the selected instance. The synthetic instances are randomly positioned between the the two minority instances.

These steps are repeated until the degree of oversampling required is attained.

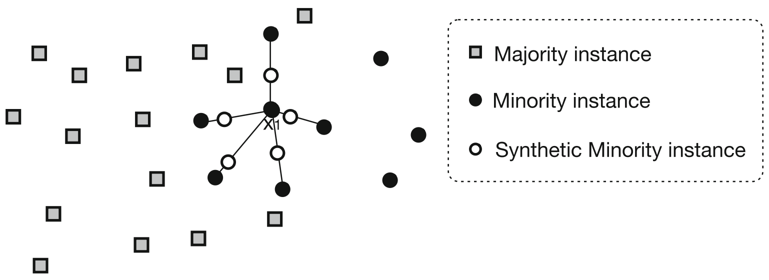

Fig. 2 How the SMOTE algorithm works (*k*=5)

3.2 ADASYN

He et al. (2008) proposed the adaptive synthetic (ADASYN) sampling approach which applies a weighted distribution method for assigning weight to different minority classes distinguishing the harder-to-classify minority samples from the easier-to-learn minority samples and adaptively shifts the classification boundary toward the difficulty minority samples. In contrast to the SMOTE algorithm which generates equal synthetic samples for each minority class, the learning algorithm is induced to focus on the hard to learn examples within the minority class samples therefore samples generated are not equal for all samples.

3.3 Borderline-SMOTE

Focusing only on the minority samples closer to the borderline, Han et al. (2005) proposed the borderline-SMOTE which generates synthetic samples along the line between some minority and majority samples based on the selected nearest neighbors. It aims at strengthening the borderline minority samples. The minority class instances are divided into three regions namely noise, borderline and safe based on the number of majority instances found among the k nearest neighbors. If all the nearest neighbors of a minority instance P are majority instances, P is denoted as noise. P is considered as a borderline instance when majority of its nearest neighbors are majority class samples. A safe P is when its neighbors are mostly minority instances. The SMOTE technique is then applied to only the borderline instances to generate more synthetic data.

3.4 Safe-level SMOTE

This is another modification of the SMOTE algorithm proposed by Bunkhumpornpat et al. (2009). A “safe-level” value is computed for each minority instance before applying the SMOTE algorithm. Synthetic samples generated are positioned closer to the largest “safe-level” ensuring that all synthetic instances are located only in the safe regions. A safe-level value is defined as the number of majority instances among the k nearest neighbors of a minority instance.

3.5 Random Over Sampling

Random Over Sampling (ROS) increases the number of minority instances by randomly replicating already existing minority class instances. As a simple and easy approach to implement, it has performed favorably well in previous empirical studies (Estabrooks et al. 2004; Kamei et al. 2007). However, no “new” data is generated since the oversampled instances are duplicates of old instances.

3.6 Random Under Sampling

Random Under Sampling (RUS) reduces the majority instances of a dataset by randomly deleting them. The minority instances are left intact while the majority instances are reduced. RUS is mostly convenient for datasets with very large number of majority samples as it reduces the training time. A major drawback of RUS is the loss of information which might be very influential to a classifier’s performance.

4 Evaluation

To conduct and report our results from the empirical study, we follow the guidelines recommended by Kitchenham et al. (2016) on performance comparison using robust statistical tests. We present the research questions and describe our experimental design and execution.

4.1 Research Questions

The main objective of this study is three-fold. To report the practical benefits of using data resampling approaches for software defect prediction research, to find a stable amount of resampling rate (*Pfp*) and to find the highest performing resampling approach to adopt for defect prediction studies. For a more comprehensive evaluation, we therefore formulate three research questions as follows:

RQ1: Is the effect of resampling approaches on defect prediction models statistically and practically significant regarding the evaluation metric and dataset used?
Prior work has examined the effect of resampling methods on performance of prediction models by considering few resampling methods, single performance measure and one-set metric datasets. Shepperd and Kadoda (2001) and Jiang et al. (2008) however, concluded that experimental conditions such as different performance measures and datasets used affect performances of prediction models. RQ1 aims to determine the significant and practical effects of applying resampling methods evaluated on two sets of different software dataset metrics, cross-versions projects and five performance measures.

RQ2: To what degree (percentage of fault-prone modules *Pfp*) should one resample or balance the original data set?
The amount of resampling (*Pfp*) applied is an important factor in determining the performance of a defect prediction model on a resampled dataset (Estabrooks et al. 2004; Weiss and Provost 2003). To maximize overall classification accuracy, most conventional prediction models assume the two classes in the datasets to be as balanced as possible or equal (Weiss and Provost 2001; Yoon and Kwek 2007). Intuitively, it may seem ideal if we stop resampling at 50%. For RQ2, we investigate the effect of applying the resampling methods at different *Pfp* rates to the training datasets with the aim of finding the most appropriate resampling rate which results in best or highest performance.

RQ3: Which are the high-performing resampling approaches for defect prediction? With the abundance of several resampling approaches proposed in literature, software quality teams face a challenge of choosing the appropriate resampling method to use. Empirically validating these approaches and finding the more stable and highest performing resampling approach has the real world benefit of helping these quality teams adopt the best resampling method to use for defect prioritization or classification. We follow up on RQ1 by conducting an overall assessment to determine and recommend the best performing resampling methods based on a robust statistical test. Thus, we apply a comparison procedure among the resampling methods for each performance measure across all twenty datasets.

4.2 Experimental Design

In this section, we present the experimental design in details comprising of the software projects and metrics considered, performance measures used for evaluating performance of prediction models, prediction models used for the experiments, statistical tests applied, parameter configurations and experimental setup.

4.2.1 Data

40 releases of 20 open source projects were used for the experiments. The datasets cover two types of metrics and have an imbalanced ratio between the ranges of 3.8 - 17.46%. This is to ensure the generalization of the experiments considering the facts that we used datasets consisting of very low and high imbalanced ratio. 10 of these projects were extracted from the PROMISE repository (Shirabad and Menzies 2005) which were collected by Jureczko and Madeyski (Jureczko and Madeyski 2010) and Jureczko and Spinellis (Jureczko and Spinellis 2010). These data have been well used in several empirical studies of defect prediction (He et al. 2012; Madeyski and Jureczko 2015; Bennin et al. 2017; Yan et al. 2017). Each project has 20 static code features (Table 1) and a labeled attribute (bug) that indicates the number of bugs for each each row or instance. An instance is non-defective if the bug value is 0; and defective otherwise.

The second half of the projects were manually extracted. For replication, publicly available projects were chosen. Additional factors considered for extracting the projects include longevity (more than a year of development) and the community of developers (more than five). Through a careful analysis of the commit logs and files from version control tools (CVS, SVN), we obtained the number of bugs in source code files for each project. For this study, we refer to a module as one source code file. By using keywords such as “error” and “bugs”, we identify modules which are defective from their commit logs. Similarly, “Bug-fixes” or “fix” keywords found in the commit logs implies that the module had been revised in the subsequent release. If files are modified for bug fixes within “X” months (timespan between version B and A) after release, we identify the files that have post-defect. For files with multiple bugs, we target all kinds of revisions, so there is no distinction between bug fixes and other revisions. Commits that were incomplete or did not contain the customized keywords were thus not included in this study. The releases chosen and the time period in which the metrics were extracted can be found in the Appendix where the “Faults recorded (Date)” column in Table 6 under Appendix A is the time period where post release bugs were recorded for each release. Refactoring for most of the projects were not performed when we extracted. As such, the value of refactorings for these projects were zero (0). During preprocessing, we excluded all metric values with zero refactorings. These projects have been used in previous studies (Bennin et al. 2016; Bennin et al. 2016). Detailed information on how the bugs were matched and the keywords used can be found in the appendix. We extracted eight typical process metrics proposed by Moser et al. (2008) and commonly used in defect prediction studies (Hata et al. 2012) from these repositories and label each module as defective or non-defective. Table 1 shows the summary of the metrics extracted while Table 2 presents the projects, number of modules and buggy labels. A module is defect-prone if the number of bugs is greater than zero. For all the data set, version B is a newer version of A.

4.2.2 Prediction Models

Five machine learning algorithms were used to construct our predictors. They include several ranges of models such as function based, instance based, tree based and kernel based models. Random Forest and the C4.5 Decision tree classifiers were selected for the tree-based models. For kernel based, the Support Vector Machines (SVM) was selected. The

Table 1 Description of the static metrics (He et al. 2012) and process metrics

Abbreviation Static Code Metrics	Description
WMC	Weighted methods per class
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling between object classes
RFC	Response for a Class
LCOM	Lack of cohesion in methods
LCOM3	Lack of cohesion in methods, different from LCOM
NPM	Number of Public Methods
DAM	Data Access Metric
MOA	Measure of Aggregation
MFA	Measure of Functional Abstraction
CAM	Cohesion Among Methods of Class
IC	Inheritance Coupling
CBM	Coupling Between Methods
AMC	Average Method Complexity
Ca	Afferent couplings
Ce	Efferent couplings
CC	McCabe's cyclomatic complexity
Max(CC)	Maximum value of CC methods of the investigated class
Avg(CC)	Arithmetic mean of the CC value in the investigated class
LOC	Lines of Code
Bug	Number of detected bugs in the class
Process Metrics	
Codechurn	Number of changed rows (number of additional lines + number of deleted rows)
LOCAdded	Additional number of rows
LOCDeleted	Deleted rows
Revisions	Revision number
Age	Last number of days since it was revised
BugFixes	Number of corrected faults
Refactorings	Number of refactoring
LOC	Number of lines of code in a module
Bugginess	Indicate the defect proneness of a module

Neural Networks model with 3-layers was also chosen as the function based classifier. Finally, the K-Nearest Neighbor model was chosen as the instance based classifier. These models were adopted because they are very common and used for several defect prediction empirical studies (Lessmann et al. 2008; Barua et al. 2014; Tang et al. 2009). The "trees"

Table 2 Summary of selected 40 Imbalanced datasets

Project Name	Metric	Version A				Version B			
		Release	Module	# Defects	Pfp(%)	Release	Module	# Defects	Pfp(%)
Ant1	Static	1.3	125	20	16	1.4	178	40	22
Ant2	Static	1.5	293	32	10.9	1.6	351	92	26.2
Camel1	Static	1.0	339	13	3.8	1.2	608	216	35.5
Camel2	Static	1.4	872	145	17	1.6	965	188	19.5
Forrest	Static	0.7	29	5	17.2	0.8	32	2	6.2
Ivy	Static	1.4	241	16	7	2.0	352	40	11.4
Jedit	Static	4.2	367	48	13.1	4.3	492	11	2.2
Synapse	Static	1.0	157	16	10.2	1.1	222	60	27
Xalan	Static	2.4	723	110	15.2	2.5	803	387	48.2
Xerces	Static	1.2	440	71	16.1	1.3	453	69	15.2
Clam Antivirus	Process	0.96	1597	93	5.82	0.97	8723	56	0.64
eCos	Process	2.0	630	110	17.46	3.0	3480	67	1.93
Helma	Process	1.3.1	312	38	12.18	1.4.0	100	17	17
NetBSD	Process	4.0	6781	548	8.08	5.0	10960	299	2.73
OpenBSD	Process	4.5	1706	275	16.12	4.6	5964	158	2.65
OpenCMS	Process	7.0.0	1727	193	11.18	8.0.0	2821	93	3.3
openNMS	Process	1.7.0	1203	123	10.22	1.8.0	1416	112	7.91
Scilab	Process	5.2.0	2636	239	9.07	5.3.0	1248	244	19.55
Spring Security	Process	2.0.0	926	126	13.61	3.0.0	639	132	20.66
XORP	Process	1.0	1696	158	9.32	1.1	1755	217	12.36

configurations of Random Forest considered for the CARET library was set from 10 up to 1000 (10,50,100, 200, 250,500,1000). The K value for the Nearest Neighbor was set to 5.

4.2.3 Experimental Setup

In emulation of a practical scenario, we train a prediction model on the previous version of a project and test on the new version (Radjenović et al. 2013; D'Ambros et al. 2010). Before the model is trained on the dataset, we first preprocess the data by applying the data sampling approaches to each training data at different *Pfp* rates. We consider five *Pfp* values of (default, 20, 30, 40, 50)% to generate 5 independent training datasets per each project. We stop at 50% because the RUS method could totally deplete datasets which have very few majority instances when applied at higher *Pfp* rates. The default value implies we apply no sampling method (NONE) to the dataset. We select these rates because all the training datasets have *Pfp* rates lower than 20%. The testing data and 5 independent training data of each project are normalized using the min-max normalization method before the model is constructed and tested. This ensures each metric value is given the same weight to help in easy classification and improvement in prediction performance. For a more robust and converging results, we run each experiment 10 times to handle the random variation and sample bias. Regarding the parameter configurations for the prediction models, a 10-fold cross-validation method is adopted during model construction to attain the best parameters to be used for testing the models on the test datasets. This method randomly partitions

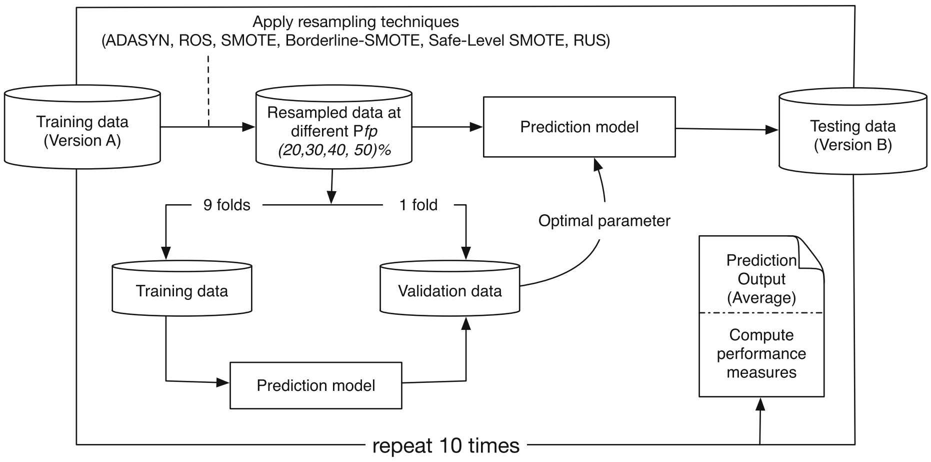
Apply resampling techniques
(ADASYN, ROS, SMOTE, Borderline-SMOTE, Safe-Level SMOTE, RUS)
Training data (Version A)
Resampled data at different P*fp* *(20,30,40, 50)%*
Prediction model
Testing data (Version B)
9 folds
1 fold
Optimal parameter
Training data
Validation data
Prediction Output (Average)
Compute performance measures
Prediction model
repeat 10 times

Fig. 3 Framework of experimental setup

the training data into 10 folds where all but one fold is used for training and validated on the remaining one fold. This is iterated until all folds have been used for training and validation. This method ensures the optimal parameter values of each model are found and used. The model construction and evaluation were conducted in R Core Team (2012), an open source statistical tool using the Caret library package (Kuhn et al. 2014). The WRS package (Wilcox and Schönbrodt 2014) was also used for the *win-tie-loss* statistical test computation and comparison. The sampling methods ADASYN, Border-line SMOTE, Safe-level SMOTE and SMOTE were implemented in MATLAB according to instructions given by their authors. These methods consider a *k*-nearest neighbor value during implementation. We used the default *k* value of 5. For the parameter configurations of our classifiers, we used the Caret library package together with the 10-fold cross-validation method in R to set the parameters for each model. The overall experimental setup is presented in Fig. 3.

4.2.4 Performance Measures for Imbalanced Datasets

Performance of most defect prediction classification models are evaluated using results computed from a confusion matrix (Table 3) where defect-prone classes are considered as positive instances and the non defect-prone classes as negative instances (Jiang et al. 2008). From Table II, the four categorized outcomes are defined as follows:

- True Positives (TP): defective modules correctly classified as defective.
- False Positives (FP): non-defective modules wrongly classified as defective.
- True Negatives (TN): non-defective modules correctly classified as not defective.
- False Negatives (FN): defective modules wrongly classified as not defective.

Table 3 Confusion Matrix for two-class problem

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

Assessing the performance of prediction models on an imbalanced dataset using the overall accuracy is inappropriate due to the unequal error cost associated with the dataset (Chawla 2010; Nickerson et al. 2001; Menzies et al. 2007). In defect prediction studies, the goal is to precisely detect the defect-prone modules, that is the minority class samples in the datasets. As such, metrics such as Precision, Recall (*pd*) and probability of false alarms (*pf*) employed by previous studies (Joshi et al. 2001; He et al. 2012; Menzies et al. 2007) for evaluating the performance of classification algorithms on minority classes are more preferable. Recall (*pd*) and *pf* was recommended by Menzies et al. (2007) as a more stable performance metric for highly imbalanced datasets. Practically, it is mostly a challenge for both recall and precision to achieve very high values simultaneously and as such F-measure which computes the harmonic mean between these two measures is preferred. We do not however use both the precision and F-measure performance measures because precision has been refuted as an unstable measure for imbalanced datasets by Menzies et al. (2007). Measures such as AUC, *balance* and *g-mean* are considered for this study. The Area Under the ROC Curve(AUC) performance measure computed from the Receiver Operating Characteristics (ROC) curve handles the trade-offs between the true and false positive error rates (Bradley 1997; Lee 2000), thus does not sacrifice one class over the other. ROC is a graphical evaluation method which aids in defect classification and prioritization. The best predictor is achieved when *pd*=1 and *pf*=0 on the ROC curve implying that all defects were actually detected with no errors. The AUC is a single value between 0 and 1. Another practical measure for software quality teams is *balance* (Menzies et al. 2007). This measure tries to find a balance between *pd* and *pf*. The Euclidean distance between the actual point (*pf,pd*) and preferred point (*pf*=0,*pd*=1) is the *balance*. Similarly, the Geometric mean (*g-mean*) performance measure proposed by Kubat et al. (1997) computes the balance between the two classes within the dataset. The measures (AUC, *balance* and *g-mean*) are used to evaluate the overall performance of the prediction models in our study. Higher values denote better prediction performance. From Table II, the mathematical definitions of *pd*, *pf*, *balance* and *g-mean* evaluation metrics used for this study are defined in Eqs. 1 to 4.

$$Recall(pd) = \frac{TP}{TP + FN} \tag{1}$$

$$pf = \frac{FP}{TN + FP} \tag{2}$$

$$balance(bal) = 1 - \frac{\sqrt{(0 - pf)^2 + (1 - pd)^2}}{\sqrt{2}} \tag{3}$$

$$g - mean = \sqrt{\frac{TP}{TP + FN} * \frac{TN}{TN + FP}} \tag{4}$$

4.2.5 Statistical Tests

To analyze the statistical significance of the performance of each sampling method on the prediction models, we adopt the Brunner's statistical test (Brunner et al. 2002) for pairwise comparison between the performance (i.e. AUC, *pd*, *pf*, *g-mean*, *balance*) values of predictors. This statistical test has recently been comprehensively validated and recommended by Kitchenham et al. (2016) as a more robust non-parametric alternative to the conventional rank-based statistical methods such as the Kruskal-Wallis and Mann-Whitney-Wilcoxon. A predictor in our study is a combination of prediction model and sampling method. 35 (5 models * 7 sampling methods) predictors are empirically investigated in this study.

Statistical Comparison For RQ1 and RQ3, we adopt the *win-tie-loss statistics* procedure, a prevalent evaluation procedure used in several Software Effort Estimation studies (Kocaguneli et al. 2012, 2013) to aid compare and summarize the statistical test results. To find the best performing predictor per each performance measure, we assess the overall performance of predictors by 3 counters - *wins, ties* and *loss*. Based on a pairwise comparison, the distributions of two predictors *k* and *l* are statistically compared using Brunner's test. Should they be statistically different, the *win* and *loss* counters of the two predictors are respectively updated by incrementing both the *win* counter of the best predictor and the *loss* counter of the worse predictor by 1. The *tie* counters are incremented by 1 for both predictors if there are no statistical differences between the distributions of the two predictors. Figure 4 presents the pseudocode for the computation of the *win-tie-loss* statistic. The comparisons are summarized by the total number of *losses*, *wins* and *wins-losses*. To address RQ2, we find the highest performance value per each performance metric except *pf* where we rather consider the lowest value and the respective *Pfp* associated with that performance value for each dataset. Using the *Pfp* rates as our counters initially set to zero, a *Pfp* rate is incremented by 1(win) when the highest performance value is attained on a dataset at this rate per each performance metric. We then aggregate the total wins across all the datasets. *Pfp* rates with the highest number of wins imply that better performances are obtained at this rate.

Effect Size Computation To examine the practical effect of the sampling methods on the prediction models, Cliff's method with Hochberg method proposed by Brunner et al. (2002) is conducted. This non-parametric method was recently recommended by Kitchenham et al. (2016) as a more effective test for handling ties in results. The delta value ($\widehat{\delta}$) ranges between 0 and 1. A high value denotes the results are significant and more practical. We interprete the effect sizes using the proposed magnitude labels of Kraemer and Kupfer (2006) i.e. small ($\widehat{\delta}$ <=0.112), medium (0.112> $\widehat{\delta}$ <0.428) and large ($\widehat{\delta}$ >=0.428). We implement the Brunner's statistical test and Cliff's method with Hochberg method using the WRS (Wilcox and Schönbrodt 2014) R package.

```
$win_k = 0, tie_k = 0, loss_k = 0$
$win_l = 0, tie_l = 0, loss_l = 0$
if $Brunner\text{-}test(Perf_k, Perf_l, 0.95)$ says they are the same then
   $tie_k = tie_k + 1$
   $tie_l = tie_l + 1$
else
   if better$(Perf_k, Perf_l)$ then
   $win_k = win_k + 1$
   $loss_l = loss_l + 1$
else
   $win_l = win_l + 1$
   $loss_k = loss_k + 1$
   end if
end if
```

Fig. 4 Pseudocode for win-tie-loss computation between predictors *k* and *l* with performance measure values $Perf_k$ and $Perf_l$. Note that *pf* values are negated before comparison since the lower the *pf* value, the better

5 Results and Analysis

In this section, we analyze the relationship between the prediction performances and the sampling methods across five different performance measures. We present the experimental results of applying the sampling methods at different *Pfp* values to the open source datasets by obtaining the performance of the prediction models on these resampled datasets. We further examine the statistical differences between and among the performances of the prediction models trained on all resampled datasets and the prediction models trained on the default datasets. Our intention is to report results of all sampling methods on all the datasets in Table 2. We compare the results based on the different performance measures. Based on the three research questions, we present the results in the sections below.

5.1 RQ1: Is the Effect of Resampling Approaches on Defect Prediction Models Statistically and Practically Significant Regarding the Evaluation Metric and Dataset Used?

In order to address RQ1, we examine the statistical significance and practical effects of applying sampling methods on prediction performance. The performance of each sampling method across each performance measure is compared to the performance value when no sampling method is applied which is represented as NONE. Figures 5 and 6 show the quartile charts of performance values for each sampling method and prediction model across all ten static-metric datasets and process-metric datasets. The results are visualized using

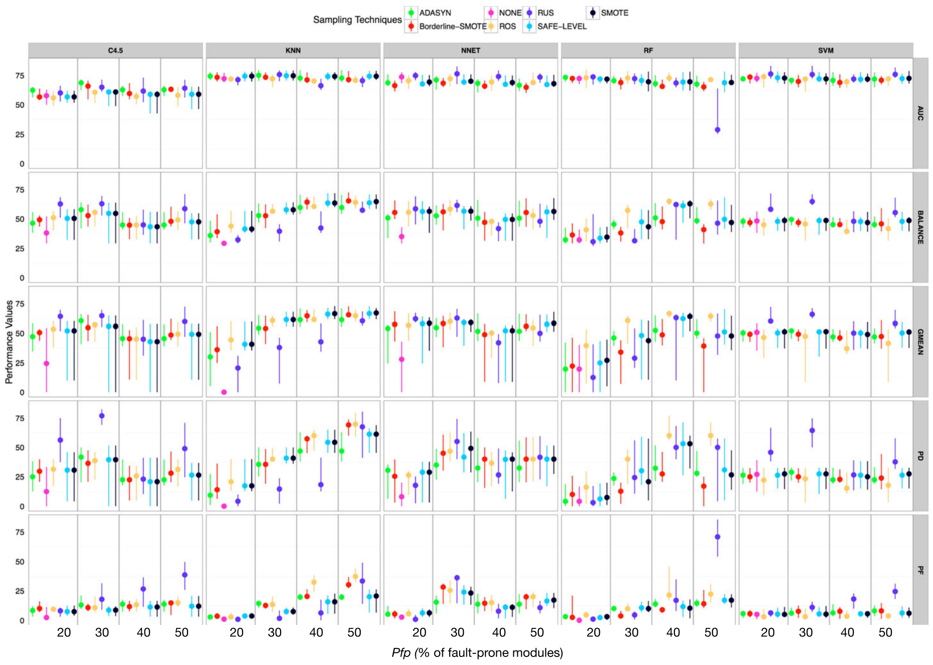

Fig. 5 Quartile charts of performance values for all sampling techniques on different Predictive Models at different (*Pfp*) values across all ten Static metrics datasets

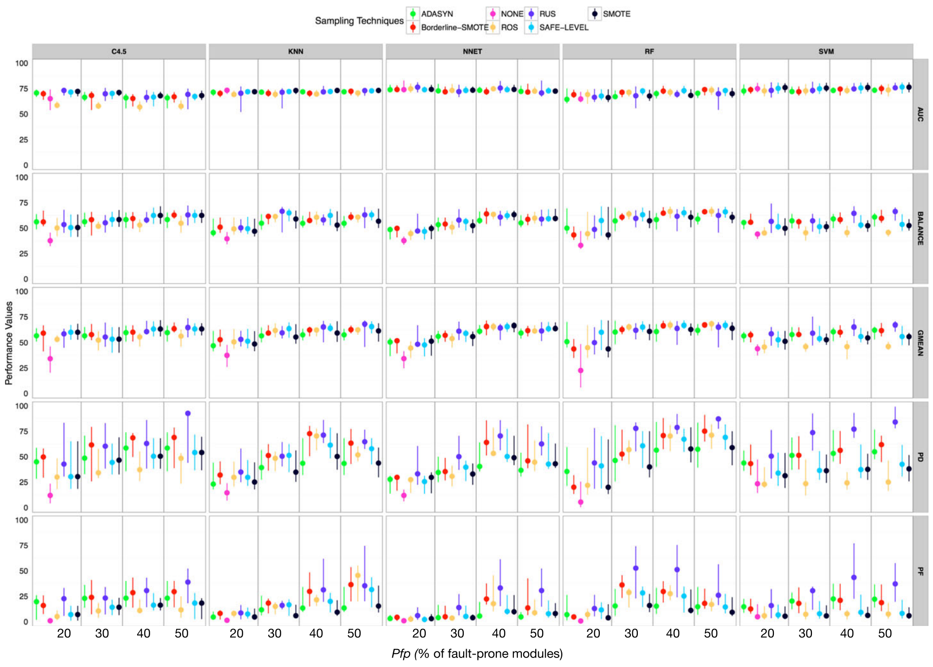

Fig. 6 Quartile charts of performance values for all sampling techniques on different Predictive Models at different (*Pfp*) values across all ten Process metrics datasets

quartile charts. Per each dataset metric type, the charts are generated for each performance measure, *Pfp* rate and prediction model by sorting the performance values for all data sets to isolate the median, lower and upper quartiles. From the performance results in Figs. 5 and 6, the median is represented by a solid dot and the 25th and 75th percentiles are represented by vertical lines.

For both dataset metric type, it is obvious sampling methods perform better compared to no sampling (NONE) across all prediction models with regards to the *balance*, *g-mean* and *pd* performance values. Using no sampling (NONE) as the baseline, most of the methods have higher values or are above the baseline. However, *pf* performance values are worse when sampling methods are applied. For the AUC performance measure, performance values seem to be no different when sampling methods are used. The *g-mean* and *balance* performance values for most sampling methods increases as *Pfp* increases. The same applies for the *pd* performance measure where the RUS method increases at an increasing rate as *Pfp* increases across all prediction models in comparison to the other sampling methods. RUS performs better in terms of *pd* because the non-defective samples consistently reduces as *Pfp* is increasing. However, this is not desirable because in as much as we want to attain high *pd* especially on the defective modules, we also do not want to lose crucial information. It is however not a surprise that as *pd* increase, *pf* also increases implying RUS should be used with extra caution as a sampling method in software defect prediction research.

We further statistically evaluate the effect of sampling methods on each defect prediction model by comparing the performance of No sampling (NONE) to the performance of applying different sampling methods to a prediction model using the *Win-Tie-Loss statistic* procedure explained in Section 4.2.5. Figures 7 and 8 presents the results in terms of wins, ties and losses generated together with the effect size magnitudes across all prediction models on the static and process metrics datasets respectively. For a record in the figure for all performance metrics except *pf*, win-tie-loss status of loss (solid orange box) and large effect size (solid green circle) embedded in the "boxes" indicates the sampling method outperformed the no sampling method and a solid green box (combination of green box representing NONE's win and green circle for large effect size) indicates otherwise. Sampling methods with more orange boxes and embedded green circles indicate better performance. From both figures, the noteworthy findings are as follows: For all performance measures except AUC and *pf*, application of different sampling methods significantly outperformed NONE. On rare occasions did the NONE outperform ROS across *balance*, *g-mean* and *pd* measures when the RF model was used. However, the performance was not practically significant as indicated by the effect size magnitude. Very few losses were recorded when no sampling method was applied to the prediction models for the AUC evaluation measure. This confirms the assertion made above that application of sampling methods does not improve AUC performance values. NONE was significantly better than all sampling methods across all models when evaluated with the *pf* performance measure. Based on the statistical tests and anlysis, we make the following conclusions from the results:

1. Performance of sampling methods are affected by the distribution of the defective and non-defective modules (*Pfp*). The performance fluctuates per the *Pfp* used for each

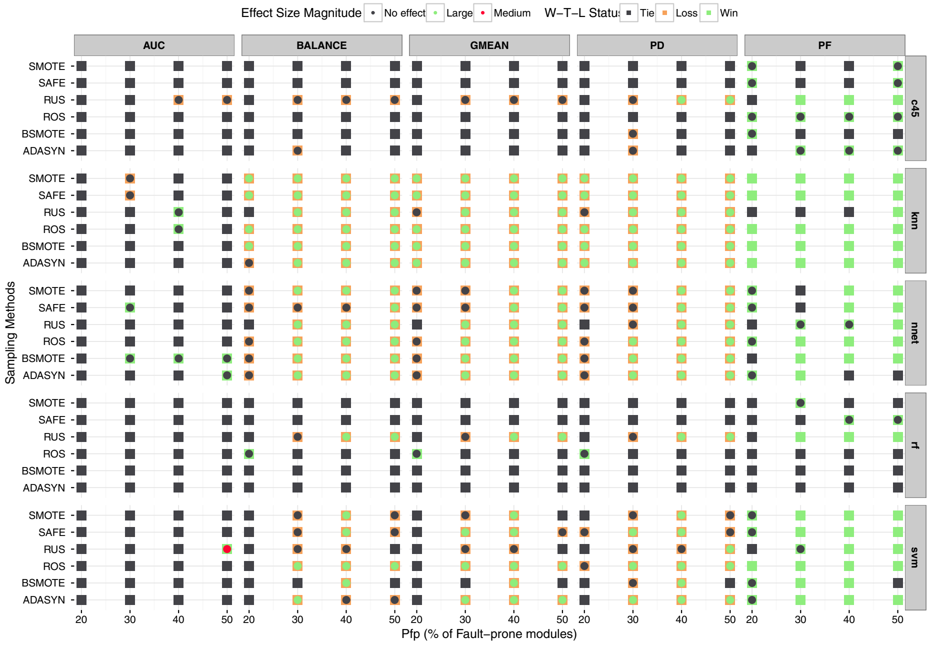

Fig. 7 Brunner's statistical test win-tie-loss comparison and effect size magnitudes of NONE vs. SMOTE, SAFE, RUS, ROS, BSMOTE and ADASYN across all 10 static metric datasets per each defect prediction model, performance measure and *Pfp* value (20%, 30%, 40%, 50%). Complete green shaded box indicate significant values for both statistical test and effect size

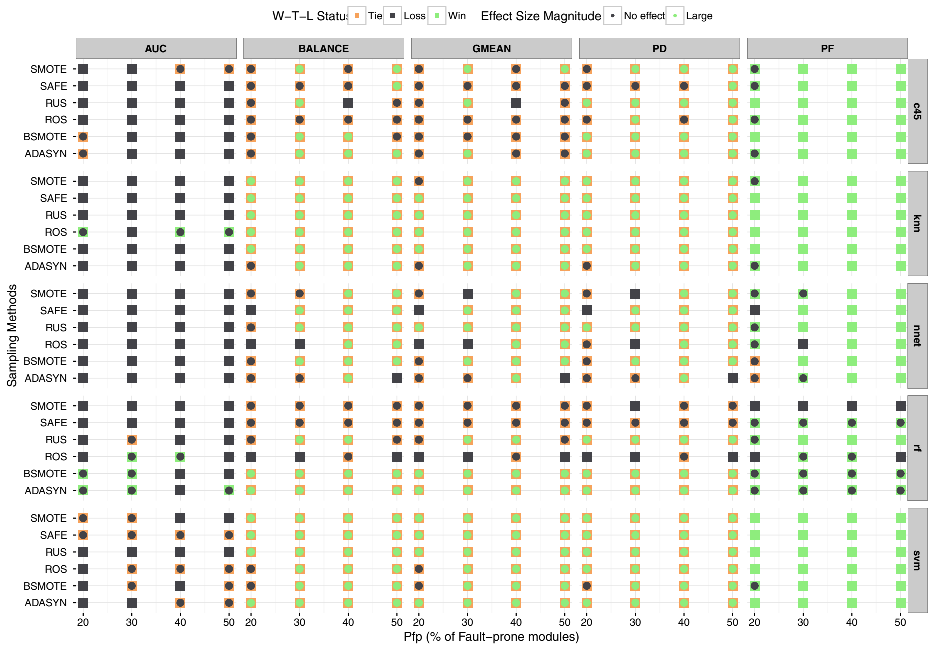

Fig. 8 Brunner's statistical test win-tie-loss comparison and effect size magnitudes of NONE vs. SMOTE, SAFE, RUS, ROS, BSMOTE and ADASYN across all 10 process metric datasets per each defect prediction model, performance measure and *Pfp* value (20%, 30%, 40%, 50%). Complete green shaded box indicate significant values for both statistical test and effect size

sampling method. No consistent or stable *Pfp* can be observed for all performance measures except *pf* measure which had best performance at a *Pfp* of 20% for all sampling methods.

2. Performance of Sampling methods are dependent on the performance measure used. Performance measures dependent on the threshold of defective/non-defective modules (that is *balance/pd/pf/g-mean*) are greatly affected by sampling methods. Sampling methods improve the *balance*, *pd* and *g-mean* values while they increase the *pf* values. On the other hand, the AUC of ROC curve performance measure which is threshold independent showed that there is no difference when sampling methods are applied to the training data.
3. Performance of sampling methods are not affected by the type of metric dataset. Similar results are observed across both static and process metric datasets.
4. Performance of sampling methods are dependent on the prediction model applied. The C4.5 prediction model was the most resistant to sampling methods with non-significant effect sizes for all performance measures except the *pf* measure.

In conclusion, the answer to **RQ1** is "yes": the effect of resampling approaches on defect prediction models are statistically and practically significant regarding the evaluation metrics considered with the exception of AUC performance measure. Additionally, performance of resampling methods are not dependent on the type of metric dataset used but rather dependent on the imbalance ratio (*Pfp*).

5.2 RQ2: To What Degree (Percentage of Fault-Prone Modules *Pfp*) Should the Original Dataset be Resampled or Balanced?

To answer RQ2, we first compute the correlation between the amount of resampling (*Pfp*) and prediction performance for each sampling method. This is done as a follow-up of validating the observed relationship between *Pfp* and prediction performance in RQ1.

Kendall's Tau correlation test between *Pfp* and prediction performance is computed. The results are presented in Table 4. For the AUC performance measure, very low and insignificant correlation values less than 0.15% was observed overall across all prediction models indicating that all the sampling methods were not affected by the *Pfp* rate. On a rare occasion did the *Pfp* have a negative and significant correlation with RUS for the RF prediction model. Positive and significant correlations between the *Pfp* rate and the remaining performance measures are observed across almost all sampling methods and prediction models. However, a positive and significant correlation for *pf* measure indicates that sampling methods do increase the probability of false alarms. This is undesirable since the lower the false alarms, the better the performance. The C4.5 and SVM prediciton models are most resistant to *Pfp* rates and are not affected by *Pfp* changes. To find the most appropriate *Pfp* rate to use per each sampling method, dot plots are presented to show the dominant *Pfp* rate that resulted in the highest performance across all datasets. More specifically, for each prediction model and sampling method, we count the number of wins (i.e., highest AUC, *g-mean*, *balance*, *pd* and lowest *pf* values) attained by each *Pfp* rate across all datasets. The results across all prediction models per each performance metric for both groups of metric datasets are displayed in Figs. 9 and 10. The best *Pfp* rates differed per the performance measure. Similar patterns were observed for both sets of metric datasets. For the AUC performance measure, the highest values are attained at a *Pfp* rate of 20% with ROS and SMOTE methods being the only exception where highest performance values were obtained consistently at a *Pfp* rate of 50%. There were no clear and consistent patterns for the *balance* and *g-mean* performance measures with the best *Pfp* alternating between 40%-50% depending on the prediction model and sampling method. Across *g-mean* and *balance* performance measures, the best *Pfp* for SMOTE and the variants (Borderline and Safe-level) was 50%. ADASYN however achieved more better performance for these measures when *Pfp* was 40%. These *Pfp* rates are however not stable since they were very close to the other *Pfp* rates. The gaps between the best *Pfp* rates and the next closest *Pfp* rates were too narrow and thus insignificant. For *pd* and *pf* performance measures, we observe more stable *Pfp* rates and consistent patterns for all sampling methods. A *Pfp* rate of 20% resulted in the lowest *pf* value across all sampling methods. In contrast, a high *Pfp* rate results in high *pd* values. A single *Pfp* rate could not be attained by ADASYN with the best ranging between 40% to 50%. From the table and figures, we draw the following conclusions:

1. There is a significant positive correlation between *Pfp* and *g-mean*, *balance* and *pd* performance measures no matter the dataset type.
2. A stable *Pfp* rate for all sampling methods is dependent on the evaluation measure employed. Performance measures dependent on the threshold of defective/non-defective modules (e.g. *balance*/*pd*/*g-mean*) achieve high or best performance values on a balanced dataset or *Pfp* rates closer to balanced datasets. The exception was *pf* where a stable *Pfp* rate of 20% was obtained. On the other hand, the AUC measure which is threshold independent had best performance for most sampling methods when *Pfp* rates closer to the default imbalance rate is applied.

Table 4 Kendall's Tau rank correlation analysis between *Pfp* and each prediction performance measure per each prediciton model and metric-type dataset

		Static Metric Datasets					
Perf	Model	ADASYN	BSMOTE	ROS	RUS	SAFE	SMOTE
AUC	c4.5	−0.06	0.08	0.01	0.13	−0.01	−0.01
	knn	−0.03	−0.07	−0.05	−0.2	0	0
	nnet	−0.11	−0.16	−0.06	0.08	−0.09	−0.04
	svm	−0.04	−0.05	−0.15	0.15	0.01	0.02
	rf	−0.13	−0.21	0.04	−0.35	−0.02	0.02
balance	c4.5	−0.02	−0.06	0	0.22	0.04	0.04
	knn	0.38	0.45	0.36	0.51	0.35	0.37
	nnet	0.13	0.21	0.3	0.39	0.17	0.21
	svm	−0.03	0	0.03	0.4	−0.02	0.01
	rf	0.27	0.06	0.41	−0.04	0.13	0.18
g-mean	c4.5	−0.04	−0.06	−0.01	0.21	0.02	0.02
	knn	0.38	0.41	0.3	0.51	0.32	0.32
	nnet	0.11	0.19	0.27	0.37	0.16	0.2
	svm	−0.03	−0.02	0.02	0.37	−0.03	0.05
	rf	0.25	0.04	0.39	0.04	0.13	0.16
pd	c4.5	−0.01	−0.06	0.05	0.41	0.05	0.05
	knn	0.4	0.56	0.49	0.55	0.45	0.45
	nnet	0.15	0.28	0.36	0.47	0.27	0.29
	svm	−0.03	−0.01	0.02	0.43	0	0
	rf	0.3	0.11	0.57	0.34	0.24	0.25
pf	c4.5	0.14	−0.04	0.19	0.44	0.08	0.08
	knn	0.49	0.73	0.64	0.48	0.38	0.38
	nnet	0.15	0.49	0.42	0.51	0.34	0.33
	svm	0.05	0.07	0.01	0.51	0.05	0.01
	rf	0.4	0.17	0.59	0.65	0.3	0.27
		Process Metric Datasets					
AUC	c4.5	−0.19	−0.05	−0.03	−0.06	0.12	0.12
	knn	0.03	−0.1	0.01	−0.12	−0.03	0.02
	nnet	−0.02	−0.03	0.07	−0.09	0.04	0.02
	svm	0.03	−0.06	0.01	−0.11	−0.06	−0.03
	rf	0.13	0.15	0.17	−0.03	0.16	0.15
balance	c4.5	0.04	−0.05	−0.01	0.02	0.14	0.13
	knn	0.13	0.17	0.13	0.13	0.26	0.22
	nnet	0.2	0.33	0.45	0.11	0.37	0.28
	svm	0.12	0.15	0.01	0.03	0.07	0.04
	rf	0.11	0.39	0.25	0.11	0.14	0.11
g-mean	c4.5	0.03	−0.06	−0.02	0.03	0.13	0.13
	knn	0.13	0.15	0.13	0.12	0.24	0.2
	nnet	0.24	0.34	0.47	0.12	0.39	0.31
	svm	0.12	0.14	0.02	0.02	0.07	0.04

Table 4 (continued)

		Static Metric Datasets					
Perf	Model	ADASYN	BSMOTE	ROS	RUS	SAFE	SMOTE
	rf	0.1	0.39	0.24	0.1	0.15	0.11
pd	c4.5	0.17	0.1	0.07	0.19	0.14	0.13
	knn	0.24	0.44	0.46	0.47	0.4	0.29
	nnet	0.2	0.44	0.47	0.37	0.46	0.32
	svm	0.17	0.19	0.02	0.32	0.08	0.03
	rf	0.19	0.5	0.39	0.28	0.29	0.22
pf	c4.5	0.13	0.13	0.15	0.18	0.18	0.15
	knn	0.31	0.52	0.53	0.4	0.45	0.33
	nnet	0.16	0.49	0.45	0.31	0.41	0.33
	svm	0.12	0.13	−0.01	0.29	0.09	0.05
	rf	0.15	0.49	0.42	0.31	0.3	0.27

Significant correlation results (p-value<0.05) are highlighted in gray

3. The prediction model used influences the *Pfp* rate to apply to the dataset for each sampling method.

From the analysis, a stable *Pfp* rate of 20% was observed for all sampling methods when evaluated with the *pf* performance measure across all prediction models. Similarly, with regards to *pd*, a stable *Pfp* rate of 50% was observed for most sampling methods. However, a stable *Pfp* rate could not be achieved for the sampling methods per the other three performance measures (AUC, g-mean, balance). It is therefore better to conduct several experiments with different *Pfp* and select the *Pfp* rate producing the highest performance value per each sampling method for these three performance measures.

5.3 RQ3: Which are the High-performing Resampling Approaches for Defect Prediction?

RQ3 aims to assess the performance of each sampling method on prediction models and determine the most effective one for defect prediction studies. To find the best sampling method, statistical comparisons are made per each performance measure using the *Pfp* rate that produced the highest performance for each sampling method and prediction model

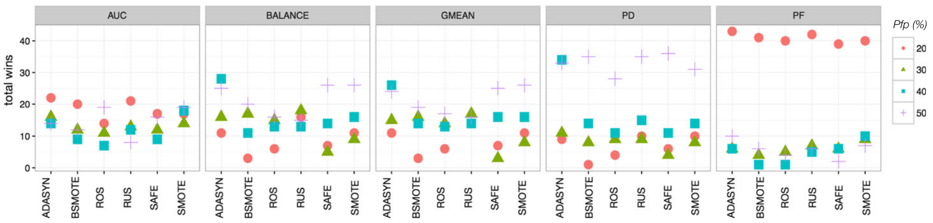

Fig. 9 Total counts of wins of each *Pfp* rate for each Sampling method across all ten process metrics datasets and all prediction models per each performance metric

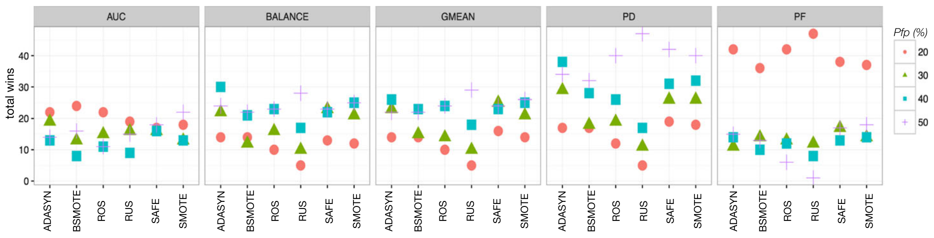

Fig. 10 Total counts of wins of each *Pfp* rate for each Sampling method across all ten static metrics datasets and all prediction models per each performance metric

across the individual datasets. To summarize the overall performance of each sampling method, we present the win-tie-loss results where each predictor is compared with 34 (7 sampling methods * 5 prediction models -1) other predictors following the statistical procedure explained in Section 4.2.5 using the best *Pfp* rate per each dataset. The results (Table 5) are then aggregated across the prediction models resulting in 170 (34*5) comparisons per each performance measure. We focus more on the wins and losses of each predictor and thus exclude the tie values from the results. From the table, ranks are ordered by the difference between wins and losses. Out of the 170 comparisons, the RUS method consistently ranked first across all performance measures except the *pf* outperforming the oversampling methods. Among the oversampling methods, Borderline-SMOTE attained the highest win-loss ratio indicating the best and stable oversampling method across all performance measures and prediction models. ADASYN and NONE consistently attained negative wins-losses ratio values for all but the *pf* performance measure indicating that prediction models perform worse when no sampling methods and ADASYN are applied to imbalanced datasets. The *win-tie-loss* results are in agreement with the results of RQ1 where sampling methods were shown to be more effective in improving performance of prediction models. However, sampling methods results in higher false alarms (*pf*). SMOTE however resulted in a positive wins-losses ratio value for *pf* whereas RUS attained the highest number of losses. Also, comparing the losses of AUC to the other performance measures, the significant improvements of sampling methods on AUC is lower. NONE attained more wins and less losses indicating sampling methods do not always improve the AUC of prediction models.

6 Discussions

6.1 General Discussion

From RQ1 and RQ2, we conclude that resampling methods do not improve AUC performance values since there were no practical significance difference between resampling methods and no sampling method. The results is in agreement with findings observed by Shatnawi (2017) that sampling methods do not improve the AUC performance of prediction models. Also, no sampling (NONE) significantly outperformed sampling methods when evaluated with the *pf* performance measure. Sampling methods significantly improved the performance (*g-mean*, *pd*, *balance*) of all prediction models. The improvements are however accompanied with a high cost of false alarms (*pf*). With regards to the best and more stable *Pfp* rate, the experimental results indicate that it is difficult to attain a stable *Pfp* to

Table 5 Performance in Terms of wins, losses, and wins-losses aggregated across all prediction models and 20 datasets per each performance measure

AUC					*g-mean*				
#	Sampling	Wins	Losses	Wins-Losses	#	Sampling	Wins	Losses	Wins-Losses
1	RUS	63	4	59	1	RUS	61	3	58
2	SMOTE	30	29	1	2	BORDERLINE	51	20	31
3	BORDERLINE	29	29	0	3	SAFE-LEVEL	58	28	30
4	SAFE-LEVEL	31	31	0	4	ROS	65	40	25
5	ADASYN	24	27	−3	5	SMOTE	44	29	15
6	ROS	30	41	−11	6	ADASYN	35	45	−10
7	NONE	19	65	−46	7	NONE	5	154	−149
pd					*pf*				
1	RUS	143	0	143	1	NONE	94	6	88
2	ROS	87	51	36	2	SMOTE	30	20	10
3	BORDERLINE	69	43	26	3	SAFE-LEVEL	16	23	−7
4	SAFE-LEVEL	56	60	−4	4	ROS	11	25	−14
5	SMOTE	41	62	−21	5	BORDERLINE	19	35	−16
6	ADASYN	34	65	−31	6	ADASYN	17	34	−17
7	NONE	5	154	−149	7	RUS	10	54	−44
balance									
1	RUS	63	3	60					
2	BORDERLINE	52	18	34					
3	ROS	67	38	29					
4	SAFE-LEVEL	53	29	24					
5	SMOTE	45	30	15					
6	ADASYN	34	47	−13					
7	NONE	4	153	−149					

Higher wins and wins-losses indicate higher performance, where as lower losses indicate higher performance. The ranks are ordered by wins-losses

be used during sampling for the AUC measure. Sampling with a *Pfp* rate of 50% produces higher *pd* values whereas a *Pfp* rate of 20% produces less *pf* values. From our stability ranking experiments in RQ3, RUS sampling method attained the highest rank across four out of five performance measures. Additionally, the Borderline-SMOTE outperformed the other oversampling methods. By conducting a comprehensive empirical experiments with more performance measures and robust statistical tests, we discuss in detail the overall findings from our analysis.

Finding 1 The *Pfp* rate applied to resampling methods significantly impacts performance of prediction models. There was a positive relationship between *Pfp* and *g-mean*, *balance* and *pd* performance measures. A negative relationship was observed between *Pfp* and *pf*. This implies prediction performance is affected by the amount of minority samples. Most machine learning algorithms have been intuitively made to favor an equally balanced dataset for the best prediction performance. The effect of *Pfp* on prediction performance is related to the information present in the dataset since the increase of minority instances tend to increase the information within the minority instances. Prediction models are known to work well when the amount of information present in the dataset is sufficient. Class imbalance due to rare minority instances or small sample size makes it difficult for learning algorithms of classifiers to generalize inductive rules (He and Garcia 2009). Menzies et al. (2008) noted that the information content within the training data is improved when sampling methods are applied and we confirm their assertion based on our experimental results. Oversampling methods aim to increase the defective information within a dataset. In contrast, undersampling methods aim to reduce non-defective information present in a dataset. Accordingly, both type of sampling methods ensure that the defective modules dominate which subsequently shifts the decision boundary of the classifier toward the region of the minority (defective) class. Synthetic oversampling methods introduce new minority instances into the dataset that provides new information to the classifier. Interestingly, simple methods such as ROS and RUS outperformed more complex and synthetic generating methods. Both methods do not provide "new" information to the classifier. However, they modify the distribution of the data which tends to positively improve performance of classifiers. ROS which tend to randomly duplicate the minority instances could duplicate instances which are very close to the decision boundary. This automatically shifts the decision boundary of any classifier trained on the dataset hence increasing the performance on the classifier on the minority instances. Similarly, RUS could randomly delete more majority instances which are close to the decision boundary thus shifting the bias of the decision boundary toward the minority instances.

Finding 2 Intrinsic algorithm make-up of sampling methods impacts prediction performance. Considering the oversampling methods, the assessment based on the *win-tie-loss* reveals that the strategy adopted by the sampling methods in generating new samples does impact prediction performance. Specifically, the individual minority data instances selected for oversampling can significantly improve prediction performance. The Borderline-SMOTE sampling method proved to be more stable across all performance measures. This method consistently ranked in the top three sampling methods for all performance measures except the *pf* according to the results in Table 5. Additionally, the Safe-Level SMOTE outperformed the original SMOTE method. As an upgrade on SMOTE, both Borderline and Safe-Level SMOTE make significant modifications to the original SMOTE sampling method. These methods specifically focus on subsets of minority instances and applies the SMOTE method to generate more samples. This finding suggest the importance of analyzing the minority instances and selecting the more relevant instances that will generate informative instances and thus provide more information to the classifier.

6.2 Sampling Methods and False Alarms (*pf*)

Sampling methods significantly increased the false alarms with RUS and ADASYN being the worse performing sampling methods. RUS method deletes majority or non-defective modules as *Pfp* increases thus information on the majority modules provided to the training model is limited or even non-existent. RUS does not provide any "new" minority instance information to classifiers similar to ROS which provides duplicated information on the minority instances. RUS and ROS make the samples limited in information leading to the rules formed by the learning algorithms to be too specific. This causes overfitting as reported by Phung et al. (2009) and thus increases the probability of false alarms. With a significant loss of information, the trained classifier lacks information on the non-defective modules and will accordingly classify non-defective modules as defective.

For the synthetic oversampling methods, the high false alarms could be attributed to the intrinsic characteristics of the algorithms specifically the selection of parent modules to used for new samples generation and positioning of the new samples generated. Most of the algorithms of synthetic oversampling methods focus on modules (parents) very close to decision boundary for new modules generation. Consequently, the positioning or placement of the new samples generated are usually ignored by these algorithms and these samples are randomly positioned based on the parent modules. As such, modules could be accidentally positioned in the region of the non-defective modules. Additionally, based on the parent selection criteria of these algorithms, if the minority class is intra-class imbalanced, that is, should there exist several or very similar modules within a specific cluster of the defective modules, the newly generated modules will most likely be a duplicate and will provide no new information to the prediction model.

In summary, sampling methods should not aim at basically making the minority samples dominate. Oversampling methods should consider the relevance of the new minority data being introduced into the data. This includes the selection of neighbors and parents instances used to generate the new data and the position or placement of the new data instance. Additionally, noise or outliers are inherent with defect prediction datasets (Gray et al. 2011). To significantly reduce the false alarm rates, sampling methods should consider noise detection and elimination techniques to ensure only relevant and important minority instances remain or are added to the training data.

6.3 Usefulness of Sampling Methods in Practice

The conducted analysis on the experiments revealed the superiority of RUS and Borderline-SMOTE over the other sampling methods. The performance measure used has significant impact on the outcome of the comparative predictors. This research has several practical implications for practitioners.

1. **Defect prioritization:** Software quality teams are mostly interested in prediction models that can aid in the efficient allocation of scarce testing resources. Prioritization of defective modules is thus necessary. The use of the area under the ROC curve (AUC) to assess the performance of prediction models does help in prioritizing the top-n defective modules which require more attention. The AUC performance values after applying sampling methods were insignificant irrespective of prediction model or dataset type. The empirical assessment based on win-tie-loss and effect sizes does reveal that sampling methods cannot be used for defect prioritization. The insignificant effect sizes computed from the Cliff's method with Hochberg's method for prediction models trained on resampled datasets demonstrate that sampling methods cannot improve the prediction of defect-proneness itself.
2. **Defect Classification:** The significant and practical results observed for the performance measures computed from confusion matrix (*g-mean*, *balance*, *pd* and *pf*) indicate the importance of sampling methods for defect classification. When software quality teams want to consider all defective modules for testing, sampling methods could be of great help. With very large effect sizes observed for the g-mean, balance and *pd* values, we have demonstrated that sampling methods can find all defective modules.

The sampling methods considered in this study explicitly set the threshold between the defective and non-defective modules. However, extra cost will be incurred in testing modules which may not be defective (false alarms) since significant *pf* values were also observed.

Points to Consider for Practitioners In summary, we provide simple guidelines for practitioners who desire to apply data resampling approaches for their defect prediction studies. The objective of the defect prediction study should indicate the resampling method to use. As observed, the use of RUS for defect prioritization purposes should be avoided. RUS could however be used for defect classification purposes. Practitioners should also consider the imbalance ratio of the training data before applying resampling methods. This is because the number of minority instances added to the training data has significant effect on prediction performance. Practitioners should consider tuning the *Pfp* value until the highest performance is obtained. Lastly, the no sampling method (NONE) is preferred if low *pf* values are desired.

7 Threats to Validity

The potential limitations of this empirical study is discussed in this section. *Construct and internal validity*: The experimental design approach adopted for empirical studies does have impact on the results. For a more comprehensive study, we used two versions of each software project, one for training and one for testing. This is a more practical approach mostly preferable to the splitting of a single project for both training and testing datasets for empirical experiments (Kamei et al. 2010; Bennin et al. 2016). We also adopted two different metrics to ensure a more generalized results. Both static code and process metrics have been shown to work very well and they are both easier to collect (Menzies et al. 2010; Moser et al. 2008; D'Ambros et al. 2012). We acknowledge that considering faults recorded in the commit logs from version control tools such as GIT and CVS implies that we missed out on faults not recorded on these tools. With regards to the recording of bug fixes for the process datasets, we acknowledge the difficulty in making clear the distinction between bug fixes for releases A and B. Further analysis and tools are thus required to capture all faults per each open source project correctly. The hyper parameters configuration set could also introduce bias in our results. Nevertheless, the experimental rig was repeated 10 times and we applied the 10-fold cross-validation approach for each run. This enabled us to report the average results across all run times.

External Validity We experimented over a large number of datasets with sufficient sizes. These projects are however open-source and results can not therefore be generalized for commercial projects. In comparison to other past studies in the literature, this study uses more data from more sources. With respect to the external validity of the prediction models, it is obvious that the number of prediction models considered for this study is a source of potential bias to our results. We considered only five prediction models spanning the machine learning, data mining and statistical based families which are widely used in several prediction models. Experiments with other prediction models not considered in this study is left for future study. For a generalized and reliable results, we evaluated the performances of the sampling methods on prediction models using five different measures: AUC, *g-mean*, *balance*, *pd* and *pf*. These measures are known to be robust and preferable for evaluating performance on imbalanced datasets.

Statistical Conclusion Validity To limit the impact of statistical conclusion validity, we used Brunner's Statistical test, a robust statistical test recommended for non-parametric tests. To find the practical effects of the results, Cliff's effect sizes were computed using Cliff's method with Hochberg's method which is able to handle ties in the results.

8 Conclusions

The impact of data resampling methods on prediction performance has not been studied in detail considering robust statistical tests. Additionally, the best distribution or resampling rate (percentage of fault-prone modules) to be used for resampling is unknown. This study examines the significant and practical impacts of resampling methods on prediction models, the best resampling rate (*Pfp*) that should be used per each resampling method and the evaluation measure adopted and the most stable and high performing resampling method suitable for software defect prediction studies. We conducted an extensive experiment on six sampling methods, five prediction models and five performance measures across 40 releases of 20 open-source software projects. The prediction performances of the models that use no sampling method (simple models) were compared with the prediction performances of the models which used resampling methods (advanced models). To decide whether the improvements were significant or not, Brunner's statistical tests were performed and Cliff's effect sizes were calculated. From our experiments, resampling methods significantly improved performance (*g-mean*, *balance* and *pd*) of prediction models across all datasets. Sampling methods however had no practical effects on the AUC performance values and significantly increased (worsened) the *pf* values. The RUS and Borderline-SMOTE proved to be more stable across several performance measures and prediction models. A stable *Pfp* rate of 20% results in better *pf* values whereas a *Pfp* rate of 50% produces higher *pd* values. We conclude that the performance of resampling methods are independent of the type of metric dataset. However, they are dependent on (1) the distribution of defective and non-defective modules in the dataset, (2) the performance measure adopted for evaluation and (3) the prediction model. Due to the high and significant *pf* values, data resampling methods should be not considered for defect prioritization activities but strongly recommended for defect classification. Since *pf* is very crucial in defect prediction studies, new resampling methods to be proposed should aim at reducing the *pf* by considering the distribution of a dataset, the selection of parent modules, position and placement of the new generated modules.

Acknowledgements This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (No. 11208017), and the research funds of City University of Hong Kong (No. 7004474, 7004683) and JSPS KAKENHI 17K00102.

Appendix A: Process Metrics Data used this Study

The static metric datasets are available on the PROMISE repository. The process metric datasets were manually extracted. The keywords used for matching the commits from the commit logs during the data extraction process are "errors, bugs, fix, fixes, issues, bugfixes, refactoring(s)". Below is a sample code used for the commit matching process.
[(bugs?—fix(es—ed)?)[\s:_#]*(\d+)],
[#(\d+)(issue—bug)s?[\s#-]*(\d+)show_bug\.cgi\?id=(\d+)PR:? (\d+)].

Table 6 presents the metrics and fault collection period for the extracted datasets. The process metric datasets are available at http://analytics.jpn.org/SEdata/.

Appendix B: Extra Results for RQ1

In the figures below, we present the AUC, *g-mean*, *balance*, *pd* and *pf* results for the sampling approaches across the prediction models on the 20 imbalanced datasets across the different percentage of fault-prone modules (*Pfp*) values.

Table 6 Summary of versions and collection period for the 10 OSS projects used (Refer to Section 4.2.1)

Project Name	Summary of Version A (Older)		
	Version	Metrics recorded (Date)	Faults recorded (Date)
Clam Antivirus	0.96	2009/10/02 - 2010/03/31	2010/03/31 - 2010/09/27
eCos	2.0	2002/11/21 - 2003/05/20	2003/05/20 - 2006/05/04
Helma	1.3.1	2003/03/01 - 2003/08/28	2003/08/28 - 2004/02/24
NetBSD	4.0	2007/06/22 - 2007/12/19	2007/12/19 - 2008/06/16
OpenBSD	4.5	2009/01/31 - 2009/05/01	2009/05/01 - 2009/07/30
OpenCMS	7.0.0	2007/01/06 - 2007/07/05	2007/07/05 - 2008/01/01
OpenNMS	1.7.0	2008/10/17 - 2009/01/15	2009/01/15 - 2009/04/15
Scilab	5.2.0	2009/06/18 - 2009/12/15	2009/12/15 - 2010/06/13
Spring Security	2.0.0	2007/10/17 - 2008/04/14	2008/04/14 - 2008/10/11
XORP	1.0	2004/01/10 - 2004/07/08	2004/07/08 - 2005/01/04
	Summary of Version B (New)		
Clam Antivirus	0.97	2010/08/11 - 2011/02/07	2011/02/07 - 2011/08/06
eCos	3.0	2008/10/01 - 2009/03/30	2009/03/30 - 2011/08/25
Helma	1.4.0	2003/10/04 - 2004/04/01	2004/04/01 - 2004/09/28
NetBSD	5.0	2008/10/31 - 2009/04/29	2009/04/29 - 2009/10/26
OpenBSD	4.6	2009/07/20 - 2009/10/18	2009/10/18 - 2010/01/16
OpenCMS	8.0.0	2010/11/09 - 2011/05/08	2011/05/08 - 2011/10/13
OpenNMS	1.8.0	2010/03/09 - 2010/06/07	2010/06/07 - 2010/09/05
Scilab	5.3.0	2010/06/18 - 2010/12/15	2010/12/15 - 2011/06/13
Spring Security	3.0.0	2009/06/25 - 2009/12/22	2009/12/22 - 2010/06/20
XORP	1.1	2004/10/15 - 2005/04/13	2005/04/13 - 2005/10/10

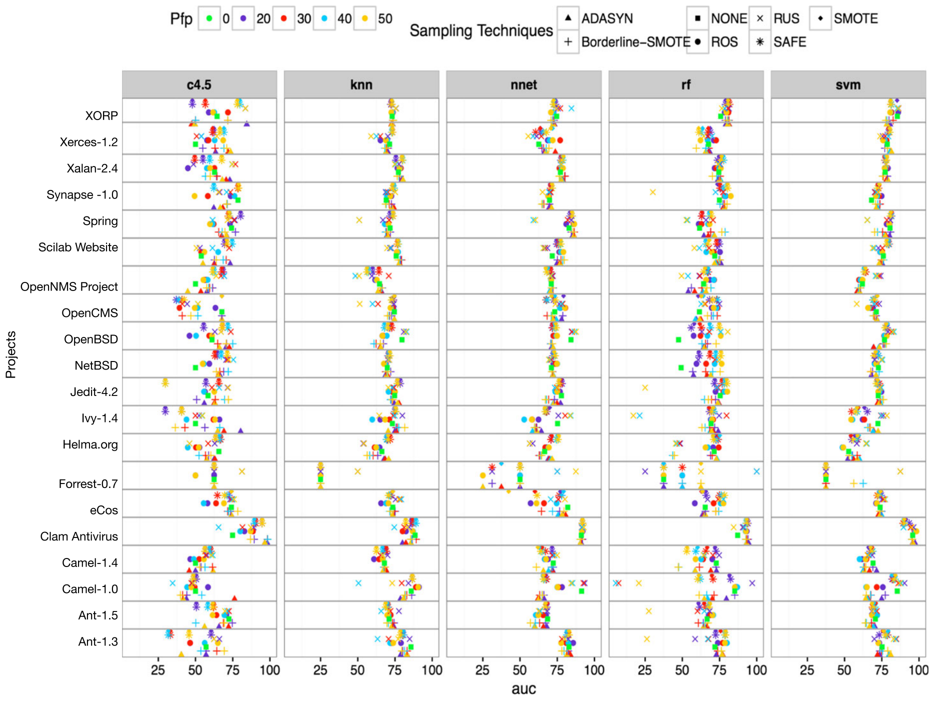

Fig. 11 Performance plot of the sampling approaches per each prediction model for the AUC performance measure on 20 imbalanced datasets across different percentage of fault-prone modules (*Pfp*) values

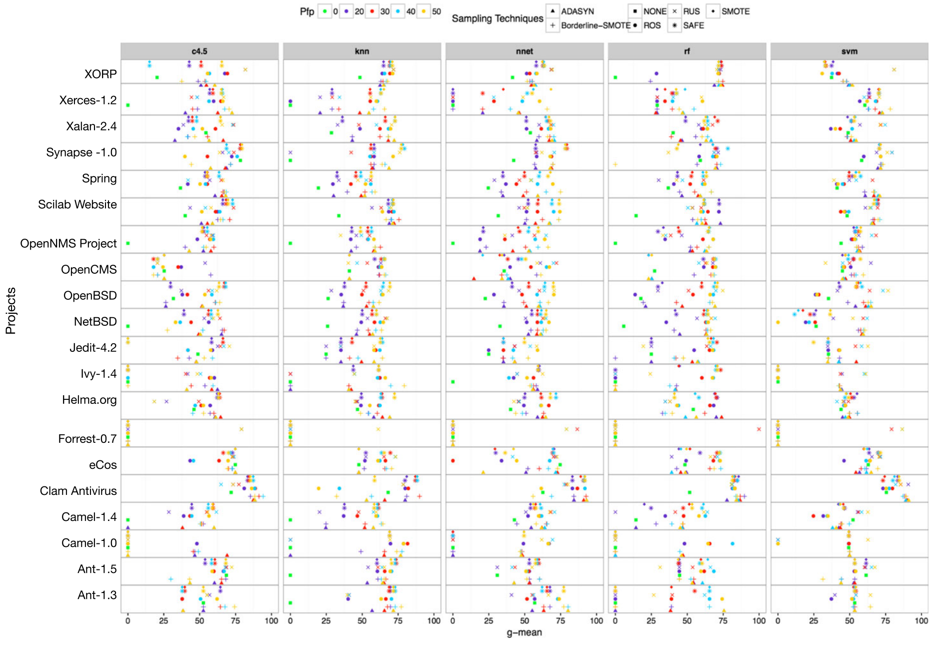

Fig. 12 Performance plot of the sampling approaches per each prediction model for the *g-mean* performance measure on 20 imbalanced datasets across different percentage of fault-prone modules (*Pfp*) values

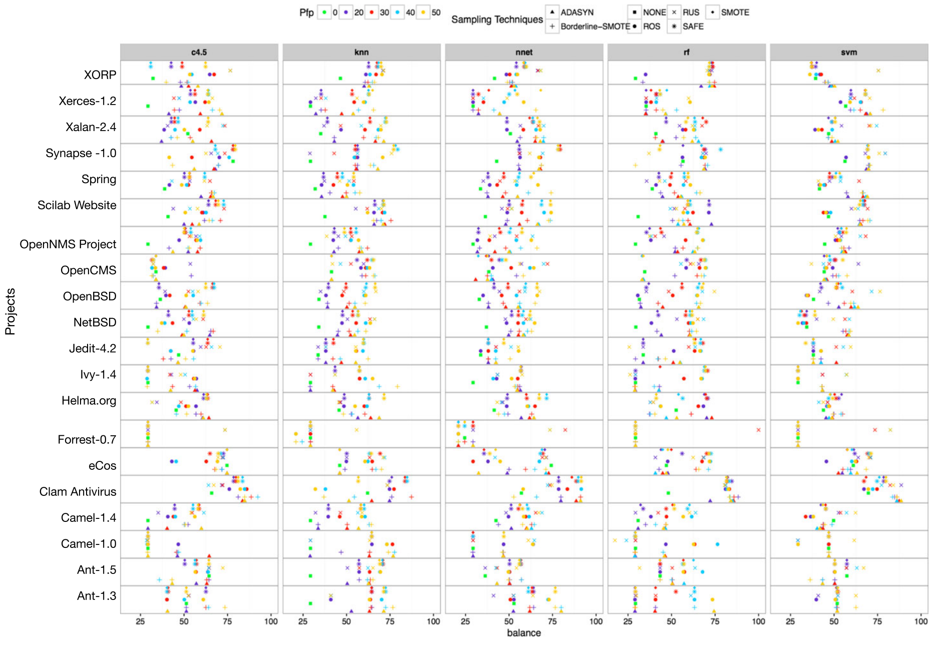

Fig. 13 Performance plot of the sampling approaches per each prediction model for the *balance* performance measure on 20 imbalanced datasets across different percentage of fault-prone modules (*Pfp*) values

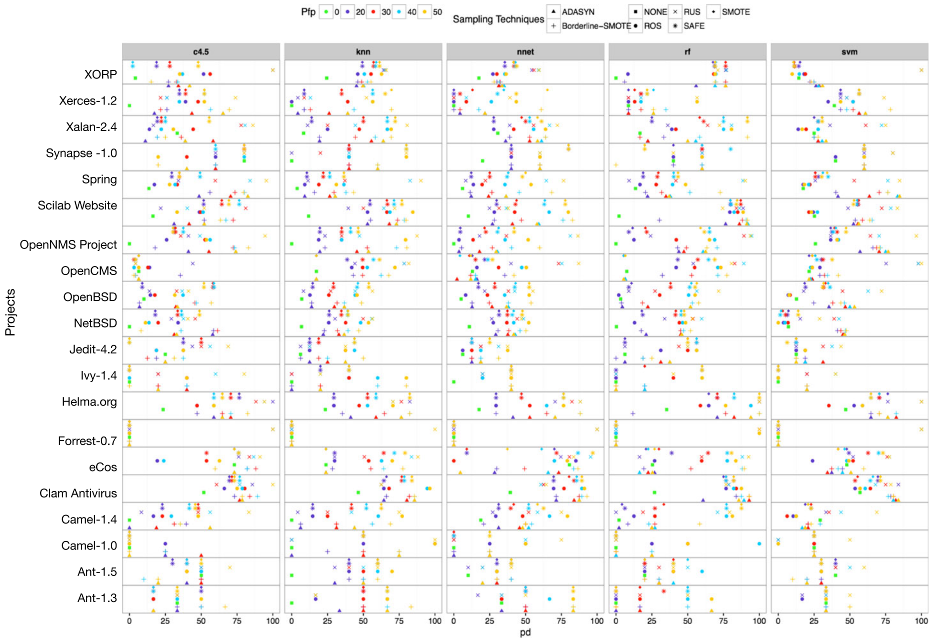

Fig. 14 Performance plot of the sampling approaches per each prediction model for the *pd* performance measure on 20 imbalanced datasets across different percentage of fault-prone modules (*Pfp*) values

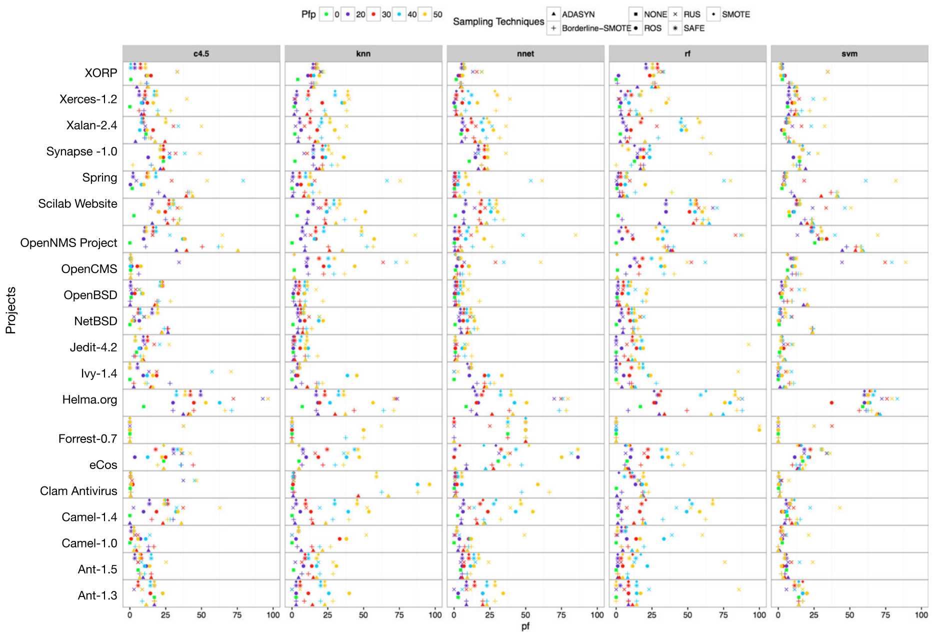

Fig. 15 Performance plot of the sampling approaches per each prediction model for the *pf* performance measure on 20 imbalanced datasets across different percentage of fault-prone modules (*Pfp*) values

References

Agrawal A, Menzies T (2017) Better data is better than better data miners (benefits of tuning smote for defect prediction). arXiv:1705.03697

Arisholm E, Briand LC, Johannessen EB (2010) A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. J Syst Softw 83(1):2–17

Barua S, Md MI, Yao Xi, Murase K (2014) Mwmote–majority weighted minority oversampling technique for imbalanced data set learning. IEEE Trans Knowl Data Eng 26(2):405–425

Bennin K, Keung J, Monden A, Phannachitta P, Mensah S (2017) The significant effects of data sampling approaches on software defect prioritization and classification. In: 11th international symposium on empirical software engineering and measurement, ESEM 2017

Bennin KE, Keung J, Monden A, Kamei Y, Ubayashi N (2016) Investigating the effects of balanced training and testing datasets on effort-aware fault prediction models. In: 2016 IEEE 40th annual Computer software and applications conference (COMPSAC), vol 1. IEEE, pp 154–163

Bennin KE, Toda K, Kamei Y, Keung J, Monden A, Ubayashi N (2016) Empirical evaluation of cross-release effort-aware defect prediction models. In: S2016 IEEE international conference on oftware quality, reliability and security (QRS). IEEE, pp 214–221

Bennin KE, Keung J, Phannachitta P, Monden A, Mensah S (2017) Mahakil: diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. IEEE Trans Softw Eng

Bradley AP (1997) The use of the area under the roc curve in the evaluation of machine learning algorithms. Pattern Recogn 30(7):1145–1159

Brunner E, Munzel U, Puri ML (2002) The multivariate nonparametric behrens–fisher problem. J Stat Plan Inference 108(1):37–53

Bunkhumpornpat C, Sinapiromsaran K, Lursinsap C (2009) Safe-level-smote: Safe-level-synthetic minority over-sampling technique for handling the class imbalanced problem. In: Pacific-asia conference on knowledge discovery and data mining. Springer, pp 475–482

Chawla NV (2010) Data mining for imbalanced datasets: an overview. In: Data mining and knowledge discovery handbook. Springer, pp 875–886

Chawla NV, Bowyer KW, Hall LO., Kegelmeyer WP (2002) Smote: synthetic minority over-sampling technique. J Artif Intell Res:321–357

D'Ambros M, Lanza M, Robbes R (2010) An extensive comparison of bug prediction approaches. In: Proceedings of 2010 7th IEEE Working Conference on Mining Software Repositories (MSR). IEEE, pp 31–41

D'Ambros M, Lanza M, Robbes R (2012) Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir Softw Eng 17(4-5):531–577

Domingos P (1999) 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, pp 155–164

Drown DJ, Khoshgoftaar TM, Seliya N (2009) Evolutionary sampling and software quality modeling of high-assurance systems. IEEE Trans Syst, Man, Cybern-Part A: Syst Humans 39(5):1097–1107

Estabrooks A, Jo T, Japkowicz N (2004) A multiple resampling method for learning from imbalanced data sets. Comput Intell 20(1):18–36

García V, Sánchez JS, Mollineda RA (2012) On the effectiveness of preprocessing methods when dealing with different levels of class imbalance. Knowl-Based Syst 25(1):13–21

Gray D, Bowes D, Davey N, Yi S, Christianson B (2011) The misuse of the nasa metrics data program data sets for automated software defect prediction. In: Proceedings of 15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011). IET, pp 96–103

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. IEEE Trans Softw Eng 38(6):1276–1304

Han H, Wang W-Y, Mao B-H (2005) Borderline-smote: a new over-sampling method in imbalanced data sets learning. In: Advances in intelligent computing. Springer, pp 878–887

Hata H, Mizuno O, Kikuno T (2012) Bug prediction based on fine-grained module histories. In: Proceedings of the 34th International Conference on Software Engineering. IEEE Press, pp 200–210

He H, Garcia EA (2009) Learning from imbalanced data. IEEE Trans knowl data Eng 21(9):1263–1284

He H, Bai Y, Garcia E, Shutao L et al (2008) Adasyn: adaptive synthetic sampling approach for imbalanced learning. In: IEEE international joint conference on Neural networks, 2008. IJCNN 2008. (IEEE world congress on computational intelligence). IEEE, pp 1322–1328

He Z, Shu F, Ye Y, Li M, Wang Q (2012) An investigation on the feasibility of cross-project defect prediction. Autom Softw Eng 19(2):167–199

Japkowicz N, Stephen S (2002) The class imbalance problem: a systematic study. Intell Data Anal 6(5):429–449

Jiang Y, Cukic B, Ma Y (2008) Techniques for evaluating fault prediction models. Empir Softw Eng 13(5):561–595

Joshi MV, Kumar V, Agarwal RC (2001) Evaluating boosting algorithms to classify rare classes: comparison and improvements. In: Proceedings IEEE International Conference on Data Mining, 2001. ICDM 2001. IEEE, pp 257–264

Jureczko M, Madeyski L (2010) Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. ACM, p 9

Jureczko M, Spinellis D (2010) Using object-oriented design metrics to predict software defects: models and methods of system dependability. Oficyna Wydawnicza Politechniki Wrocławskiej:69–81

Kamei Y, Monden A, Matsumoto S, Kakimoto T, Matsumoto Kx-I (2007) The effects of over and under sampling on fault-prone module detection. In: First international symposium on empirical software engineering and measurement, 2007. ESEM 2007. IEEE, pp 196–204

Kamei Y, Matsumoto S, Monden A, Matsumoto K-I, Adams B, Hassan AE (2010) Revisiting common bug prediction findings using effort-aware models. In: Proceedings of 2010 IEEE International Conference onSoftware Maintenance (ICSM). IEEE, pp 1–10

Kitchenham B, Madeyski L, Budgen D, Keung J, Brereton P, Charters S, Gibbs S, Pohthong A (2016) Robust statistical methods for empirical software engineering. Empir Softw Eng:1–52

Kocaguneli E, Menzies T, Bener A, Keung JW (2012) Exploiting the essential assumptions of analogy-based effort estimation. IEEE Trans Softw Eng 38(2):425–438

Kocaguneli E, Menzies T, Keung J, Cok D, Madachy R (2013) Active learning and effort estimation: finding the essential content of software effort estimation data. IEEE Trans Softw Eng 39(8):1040–1053

Kraemer HC, Kupfer DJ (2006) Size of treatment effects and their importance to clinical research and practice. Biological Psych 59(11):990–996

Kubat M, Matwin S et al (1997) Addressing the curse of imbalanced training sets: one-sided selection. In: ICML, vol 97, Nashville, USA, pp 179–186

Kuhn M, Wing J, Weston S, Williams A, Keefer C, Engelhardt A, Cooper T, Mayer Z (2014) Caret: classification and regression training. r package version 6.0–24

Laradji IH, Alshayeb M, Ghouti L (2015) Software defect prediction using ensemble learning on selected features. Inf Softw Technol 58:388–402

Lee SS (2000) Noisy replication in skewed binary classification. Comput Stat Data Anal 34(2):165–191

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans Softw Eng 34(4):485–496

Liu M, Miao L, Zhang D (2014) Two-stage cost-sensitive learning for software defect prediction. IEEE Trans Reliab 63(2):676–686

Madeyski L, Jureczko M (2015) Which process metrics can significantly improve defect prediction models? an empirical study. Softw Qual J 23(3):393–422

Menzies T, Dekhtyar A, Distefano J, Greenwald J (2007) Problems with precision: a response to comments on data mining static code attributes to learn defect predictors. IEEE Trans Softw Eng 33(9):637

Menzies T, Greenwald J, Frank A (2007) Data mining static code attributes to learn defect predictors. IEEE Trans Softw Eng 33(1):2–13

Menzies T, Turhan B, Bener A, Gay G, Cukic B, Jiang Y (2008) Implications of ceiling effects in defect predictors. In: Proceedings of the 4th international workshop on Predictor models in software engineering. ACM, pp 47–54

Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener AY (2010) Defect prediction from static code features: current results, limitations, new approaches. Autom Softw Eng 17(4):375–407

Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: ACM/IEEE 30th international conference on Software engineering, 2008. ICSE'08. IEEE, pp 181–190

Nickerson A, Japkowicz N, Milios E (2001) Using unsupervised learning to guide resampling in imbalanced data sets. In: Proceedings of the Eighth International Workshop on AI and Statitsics, pp 261–265

Pazzani M, Merz C, Murphy P, Ali K, Hume T, Brunk C (1994) Reducing misclassification costs. In: Proceedings of the Eleventh International Conference on Machine Learning, pp 217–225

Pelayo L, Dick S (2007) Applying novel resampling strategies to software defect prediction. In: Annual meeting of the north american Fuzzy information processing society, 2007. NAFIPS'07. IEEE, pp 69–72

Phung SL, Bouzerdoum A, Nguyen GH (2009) Learning pattern classification tasks with imbalanced data sets

Radjenović D, Heričko M, Torkar R, Živkovič A (2013) Software fault prediction metrics: a systematic literature review. Inf Softw Technol 55(8):1397–1418

Riquelme JC, Ruiz R, Rodríguez D, Moreno J (2008) Finding defective modules from highly unbalanced datasets. Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos 2(1):67–74

Shirabad JS, Menzies TJ (2005) The PROMISE repository of software engineering databases. School of Information Technology and Engineering, University of Ottawa, Canada

Seiffert C, Khoshgoftaar TM, Hulse JV, Rusboost AN (2010) A hybrid approach to alleviating class imbalance. IEEE Trans Syst Man, and Cybernetics-Part A: Systems and Humans 40(1):185–197

Shanab A, Khoshgoftaar TM, Wald R, Napolitano A (2012) Impact of noise and data sampling on stability of feature ranking techniques for biological datasets. In: 2012 IEEE 13th international conference on Information reuse and integration (IRI). IEEE, pp 415–422

Shatnawi R (2017) The application of roc analysis in threshold identification, data imbalance and metrics selection for software fault prediction. Innov Syst Softw Eng:1–17

Shepperd M, Kadoda G (2001) Comparing software prediction techniques using simulation. IEEE Trans Softw Eng 27(11):1014–1022

Sun Y, Kamel MS, Wong AKC, Wang Y (2007) Cost-sensitive boosting for classification of imbalanced data. Pattern Recogn 40(12):3358–3378

Sun Z, Song Q, Zhu X (2012) Using coding-based ensemble learning to improve software defect prediction. IEEE Trans Syst, Man, Cybern, Part C (Appl Rev) 42(6):1806–1817

Tang Y, Zhang Y-Q, Chawla NV, Krasser S (2009) Svms modeling for highly imbalanced classification. IEEE Trans Syst, Man, Cybern, Part B (Cybernetics) 39(1):281–288

R Core Team (2012) R: a language and environment for statistical computing. Vienna, Austria: R Foundation for Statistical Computing

Wang S, Yao X (2013) Using class imbalance learning for software defect prediction. IEEE Trans Reliab 62(2):434–443

Weiss GM, Provost F (2001) The effect of class distribution on classifier learning: an empirical study. Rutgers Univ

Weiss GM, Provost F (2003) Learning when training data are costly: the effect of class distribution on tree induction. J Artif Intell Res:315–354

Wilcox RR, Schönbrodt FD (2014) The wrs package for robust statistics in r (version 0.26). Available: Retrieved from https://github.com/nicebread/WRS

Wong GY, Leung FHF, Ling S-H (2013) A novel evolutionary preprocessing method based on over-sampling and under-sampling for imbalanced datasets. In: 2013-39th annual conference of the IEEE Industrial electronics society, IECON. IEEE, pp 2354–2359

Yan M, Fang Y, Lo D, Xia X, Zhang X (2017) File-level defect prediction: unsupervised vs. supervised models. In: 2017 ACM/IEEE international symposium on Empirical software engineering and measurement (ESEM). IEEE, pp 344–353

Yoon K, Kwek S (2007) A data reduction approach for resolving the imbalanced data issue in functional genomics. Neural Comput Appl 16(3):295–306

Zheng J (2010) Cost-sensitive boosting neural networks for software defect prediction. Expert Syst Appl 37(6):4537–4543

Kwabena Ebo Bennin is a PhD candidate and HKPFS Fellow at the Department of Computer Science, City University of Hong Kong. He is under the supervision of Dr. Jacky Keung. He was a visiting researcher at Okayama University, Japan working under the supervision of Professor Akito Monden. He received his Bachelor degree (Hons) in Computer Science and Statistics from University of Ghana in 2011. His Ph.D. thesis aims to improve the fundamentals of predictive modeling for software engineering (i.e. defect prediction models) in order to produce more accurate predictions and reliable insights. His research interests also include data mining for software engineering datasets, software effort and cost estimation and Proposing techniques for improving bugs localization.

Jacky W. Keung received the B.Sc. (Hons.) in Computer Science from the University of Sydney, and the Ph.D. in Software Engineering from the University of New South Wales, Australia. He is Assistant Professor in the Department of Computer Science, City University of Hong Kong. His main research area is in software effort and cost estimation, empirical modelling and evaluation of complex systems, and intensive data mining for software engineering datasets. His research has been published in prestigious journals including IEEE Transactions on Software Engineering, Empirical Software Engineering Journal, Information and Software Technology, the Journal of Systems and Software, Automated Software Engineering and many other leading journals and conferences.

Dr. Keung is also the vice-president of IEEE Hong Kong Section Computer Society Chapter.

Akito Monden is a professor in the Graduate School of Natural Science and Technology at Okayama University, Japan. He received the BE degree (1994) in electrical engineering from Nagoya University, and the ME (1996) and DE (1998) degrees in information science from Nara Institute of Science and Technology (NAIST). His research interests include software measurement and analytics, and software security and protection. He is a member of the IEEE, ACM, IEICE, IPSJ and JSSST.