A Comprehensive Investigation of the Role of Imbalanced Learning for Software Defect Prediction

Qinbao Song, Yuchen Guo and Martin Shepperd*

This article is dedicated to the memory of Prof. Qinbao Song (1966-2016)

Abstract—***Context***: Software defect prediction (SDP) is an important challenge in the field of software engineering, hence much research work has been conducted, most notably through the use of machine learning algorithms. However, class-imbalance typified by few defective components and many non-defective ones is a common occurrence causing difficulties for these methods. Imbalanced learning aims to deal with this problem and has recently been deployed by some researchers, unfortunately with inconsistent results.
Objective: We conduct a comprehensive experiment to explore (a) the basic characteristics of this problem; (b) the effect of imbalanced learning and its interactions with (i) data imbalance, (ii) type of classifier, (iii) input metrics and (iv) imbalanced learning method.
Method: We systematically evaluate 27 data sets, 7 classifiers, 7 types of input metrics and 17 imbalanced learning methods (including doing nothing) using an experimental design that enables exploration of interactions between these factors and individual imbalanced learning algorithms. This yields 27x7x7x17 = 22491 results. The Matthews correlation coefficient (MCC) is used as an unbiased performance measure (unlike the more widely used F1 and AUC measures).
Results: (a) we found a large majority (87%) of 106 public domain data sets exhibit moderate or low level of imbalance (imbalance ratio $<$10; median = 3.94); (b) Anything other than low levels of imbalance clearly harm the performance of traditional learning for SDP; (c) imbalanced learning is more effective on the data sets with moderate or higher imbalance, however negative results are always possible; (d) type of classifier has most impact on the improvement in classification performance followed by the imbalanced learning method itself. Type of input metrics is not influential. (e) only $\sim 52\%$ of the combinations of Imbalanced Learner and Classifier have a significant positive effect.
Conclusion: This paper offers two practical guidelines. First, imbalanced learning should only be considered for moderate or highly imbalanced SDP data sets. Second, the appropriate combination of imbalanced method and classifier needs to be carefully chosen to ameliorate the imbalanced learning problem for SDP. In contrast, the indiscriminate application of imbalanced learning can be harmful.

Index Terms—Software defect prediction, bug prediction, imbalanced learning, imbalance ratio, effect size.

1 Introduction

To help ensure software quality, much effort has been invested on software module testing, yet with limited resources this is increasingly being challenged by the growth in the size and complexity of software systems. Effective defect prediction could help test managers locate bugs and allocate testing resources more efficiently, thus it has become an extremely popular research topic [1], [2].

Obviously this is an attractive proposition, however despite a significant amount of research, it is having limited impact upon professional practice. One reason is that researchers are presenting mixed signals due to the inconsistency of results (something we will demonstrate in our summary review of related defect prediction experiments in Section 2.2). We aim to address this via attention to the relationship between data set and predictor, secondly by integrating all our analysis into a single consistent and comprehensive experimental framework, and thirdly by avoiding biased measures of prediction performance. So our goal is to generate conclusions that are actionable by software engineers.

Machine learning is the dominant approach to software defect prediction [3]. It is based on historical software information, such as source code edit logs [4], bug reports [5] and interactions between developers [6]. Such data are used to predict which components are more likely to be defect-prone in the future. We focus on the classification based methods since these are most commonly used. These methods first learn a classifier as the predictor by applying a specific algorithm to training data, then the predictor is evaluated on new unseen software module as a way to estimate its performance if it were to be used in the 'wild' using cross-validation.

A problem frequently encountered is that real world software defect data consists of only a few defective components (usually referred to as positive cases) and a large number of non-defective ones (negative cases) [7]. Consequently the distribution of software defect data is highly

- *Q. Song and Y. Guo are with the Dept. of Computer Science & Technology, Xi'an Jiaotong University, China. E-mail: wispcat@stu.xjtu.edu.cn*
  Q. Song is also with the State Key Laboratory of Software Engineering at Wuhan University, China.
- *M. Shepperd is with the Dept. of Computer Science, Brunel University London, UK. E-mail: martin.shepperd@brunel.ac.uk*
- **corresponding author*

Manuscript received June 13, 2017; revised January 20, 2018; revised April 27, 2018.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

skewed, known as class imbalanced data in the field of machine learning. When learning from class imbalanced data, traditional machine learning algorithms struggle [8] and consequently perform poorly in finding rare classes. The underlying reasons are that most algorithms:

- assume balanced class distributions or equal misclassification costs [9], thus fail to properly represent the distributive characteristics of the imbalanced data.
- are frequently designed, tested, and optimized according to biased performance measures that work against the minority class [10], [11]. For example, in the case of accuracy, a trivial classifier can predict all instances as the majority class, yielding a very high accuracy rate yet with no classification capacity.
- utilize a bias that encourages generalization and simple models to avoid the possibility of over-fitting the underlying data [12]. However, this bias does not work well when generalizing small disjunctive concepts for the minority class [13]. The learning algorithms tend to be overwhelmed by the majority class and ignore the minority class [14], a little like finding proverbial needles in a haystack.

As a result, imbalanced learning has become an active research topic [9], [8], [15] and a number of imbalanced learning methods have been proposed such as bagging [16], boosting [17] and SMOTE [18]. Imbalanced learning has also drawn the attention of researchers in software defect prediction. Yet, although imbalanced learning *can* improve prediction performance, overall the results seem to be quite mixed and inconsistent.

We believe there are three main reasons for this uncertainty concerning the use of imbalanced learning for software defect prediction. First, commonly used performance measures are biased. Second, the imbalance level of software defect data and its the relationship with the predictive performance are unexplored. Third, the interaction between the choice of imbalanced learning methods and choice of classifiers is not well understood. Likewise with the choice of data set and input metric types (e.g., static code or process metrics, network metrics). Fourth, different studies employ differing experimental procedures, choice of hyper-parameters, etc which may render results not strictly comparable. As [11] report, differences between research group are a major source of variance in experimental results.

Consequently, there is a need to systematically explore the research questions below:

1) How imbalanced are software defect data sets?
2) How does traditional learning perform under imbalanced data?
3) How does imbalanced learning perform compared with traditional learning?
4) What are the interactions between: (i) data sets (including imbalance ratio[1] and types of input metric) (ii) type of classifier (iii) type of imbalanced learning method?

1. The class imbalance ratio (IR) is defined as non-defective (majority) relative to defective (minority). Generally, $IR = \frac{Major}{Minor}$ [19] but in software defect prediction, we assume the defective (positive) is the minor class.

This paper makes the following contributions:

1) Given the complexity and contradictory nature of results emerging from other studies, we exhaustively evaluate the impact of different learners and data sets. We believe this to be the largest single experimental investigation of imbalanced learning for software defect prediction as we evaluate the performance of 16 imbalanced methods plus a benchmark of a null imbalanced method making a total of 17 approaches which are combined with 7 examples of the main types of classifiers and 7 classes of input metric, yields 27x7x7x17 = 22491 results.
2) We quantify and categorize the degree of imbalance in all publicly available software defect data (106 data sets).
3) We generate a number of practical or actionable findings. We show that imbalanced data is a challenge for software defect prediction. Our findings suggest that imbalanced learners should be deployed if the imbalance level is not low. We show that the blind application of imbalanced learners are not automatically successful, but that particular combinations of imbalance learner and classifier can yield very practical improvements in prediction.
4) We demonstrate that typical classification performance measures (e.g., *F-measure* and *Area under the Curve (AUC)*) are unsound and demonstrate a practical alternative in the form of the Matthews correlation coefficient (MCC). We also focus on *effect size* namely dominance rather than p-values.
5) Our R code and data are shared via zenodo at https://zenodo.org/badge/latestdoi/94171384

The remainder of this paper is organized as follows. Section 2 provides a brief introduction to imbalanced learning methods and summarizes how these ideas have been applied in software defect prediction research. It then shows that many results are inconsistent. Section 3 sets out the details of our experimental design and the data used. Next, Section 4 presents and discusses our experimental results. Section 5 considers potential threats to validity and our mitigating actions; Section 6 draws our study conclusions.

2 Related Work

2.1 Imbalanced Learning

A good deal of work has been carried out by the machine learning community — although less so in empirical software engineering — to solve the problem of learning from imbalanced data. Imbalanced learning algorithms can be grouped into five categories:

- Sub-Sampling
- Cost-Sensitive Learning
- Ensemble Learning
- Imbalanced Ensemble Learning
- Special-purpose Learning

We briefly review these. For more details see [9], [20], [21].

Sub-sampling is a data-level strategy in which the data distribution is re-balanced prior to the model construction so that the learned classifiers can perform in a similar way

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

to traditional classification [22], [18]. Within sub-sampling there are four main approaches. *1) Under-sampling* extracts a subset of the original data by the random elimination of majority class instances, but the major drawback is that this can discard potentially useful data. *2) Over-sampling* creates a superset of the original data through the random replication of some minority class instances, however, this may increase the likelihood of overfitting [20]. *3) SMOTE* [18] is a special over-sampling method that seeks to avoid overfitting by synthetically creating new minority class instances by means of interpolation between near neighbours . *4) Hybrid* methods combine more than one sub-sampling technique [23].

Cost-sensitive learning can be naturally applied to address imbalanced learning problems [24]. In the context of defect prediction, false negatives are likely to be considerably more costly than false positives. Instead of balancing data distributions through sub-sampling, cost-sensitive learning optimizes training data with a cost matrix that defines the different misclassification costs for each class. A number of cost-sensitive learning methods have been developed by using cost matrices, such as cost-sensitive K-nearest neighbors [25], cost-sensitive decision trees [26], cost-sensitive neural networks [27], and cost-sensitive support vector machines [28]. Unfortunately misclassification costs are seldom available[2].

Ensemble learning is the basis of generalizability enhancement; each classifier is known to make errors, but different classifiers have been trained on different data, so the corresponding misclassified instances are not necessarily the same [30]. The most widely used methods are Bagging [16] and Boosting [17] whose applications in various classification problems have led to significant improvements [31]. Bagging consists of building different classifiers with bootstrapped replicas of the original training data. Boosting serially trains each classifier with the data obtained by weighted sampling from the original data, in order to focus on difficult instances. AdaBoost [17] is the most widely used boosting method, and was identified as one of the top ten most influential data mining algorithms [32].

Imbalanced ensemble learning combines ensemble learning with the aforementioned sub-sampling techniques to address the problems of imbalanced data classification. The idea is to embed a data preprocessing technique into an ensemble learning method to create an imbalanced ensemble learner. For instance, if under-sampling, over-sampling, underover-sampling, and SMOTE rather than the standard random sampling used by Bagging, were carried out before training each classifier this leads to UnderBagging [14], OverBagging [33], UnderOverBagging [14], and SMOTEBagging [14]. In the same way, by integrating under-sampling and SMOTE with Boosting we obtain RUSBoost [34] and SMOTEBoost [35]. Unlike sampling-based ensemble methods, EM1v1 [36] handles the imbalanced data with splitting and coding techniques.

Special-purpose Learning is specialized to a particular type of classifier or existing algorithm, e.g., kernel-based methods for SVM [9]. Since this type of imbalanced learning by definition cannot be applied to each type of classifier this does not fit our experimental design and is excluded. For more details see [21].

2.2 Software Defect Prediction

As discussed, researchers are actively seeking means of predicting the defect-prone components within a software system. The majority of approaches use historical data to induce prediction systems, typically dichotomous classifiers where the classes are defect or not defect-prone. Unfortunately software defect data are highly prone to the class-imbalance problem [38], yet "many studies [still] seem to lack awareness of the need to account for data imbalance" [1]. Fortunately there have been a number of recent experiments that explicitly address this problem for software defect prediction.

Table 1 summarizes this existing research. Defect prediction methods can be viewed as a combination of classification algorithm, imbalanced learning method and class of input metric. We highlight seven different classifier types (C4.5, ..., NB) in conjunction with 16 different imbalanced learners (Bag, ..., SBst) together with the option of no imbalanced learning yielding 17 possibilities. Method labels are constructed as `<classifier> + <imbalanced learner>` for instance `NB+SMOTE` denotes Naïve Bayes coupled with SMOTE. Next there are four[3] classes of metric (code, ... code+network+process) yielding $7 \times 17 \times 4 = 476$ combinations displayed and a further 357 implicit combinations.

Each cell in Table 1 denotes published experiments that have explored a particular interaction. Note that the matrix is relatively sparse with only 54 cells covered ($54/833 \approx 6\%$) indicating most combinations have yet to be explored. This is important because it is quite possible that there are interactions between the imbalanced learner, classifier and input metrics such that it may be unwise to claim that a particular imbalanced learner has superior performance, when it has only been evaluated on a few classifiers. Indeed some types of input metric e.g., code + network metrics have yet to be explored in terms of unbalanced learning. By contrast, five independent studies have explored the classifier C4.5 with under-sampling.

Furthermore, some of these experiments report conflicting results. The underlying reasons include differing data sets, experimental design and performance measures along with differing parameterization approaches for the classifiers [11]. This makes it very hard to determine what to conclude and what advice to give practitioners seeking to predict defect-prone software components. We give three examples of conflicting results.

First, Menzies et al. [39] conducted an experiment based on 12 PROMISE data sets. Their results showed that sub-sampling offers no improvement over unsampled Naïve Bayes which does outperform sub-sampling C4.5. This is confirmed by Sun et al. [36]. However, Menzies et al. also

2. Misclassification costs could be given by domain experts, or can be learned via other approaches [29], but do not naturally exist. Typically, the cost of misclassifying minority instances is higher than the opposite, which biases classifiers toward the minority class.

3. Strictly speaking there are seven combinations of metric class however, Network, Process and Network+Process are all empty i.e., thus far unexplored, so for reasons of space they are excluded from Table 1.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

Method	Metrics				Method	Metrics				Method	Metrics			
	Code	Code +Network	Code +Process	Code +Network +Process		Code	Code +Network	Code +Process	Code +Network +Process		Code	Code +Network	Code +Process	Code +Network +Process
C4.5	[37][36][38][39][40][41][42]		[4][43][42]		SVM+Bst					IBk+OBag				
RF	[36][7][40]				SVM+US	[40]				IBk+UOBag				
SVM	[40]				SVM+OS	[40]				IBk+SBag				
Ripper	[36][40]				SVM+UOS					IBk+UBst				
Ibk	[40]				SVM+SMOTE	[40]				IBk+OBst				
LR	[44][40]		[4]		SVM+COS					IBk+UOBst				
NB	[36][7][39][40][42]		[4][42]		SVM+EM1v1					IBk+SBst				
C4.5+Bag	[36]				SVM+UBag					LR+Bag				
C4.5+Bst	[7][37][36]		[43]		SVM+OBag					LR+Bst				
C4.5+US	[37][36][7][39][40]				SVM+UOBag					LR+US	[44][40]			
C4.5+OS	[37][36][39][40]				SVM+SBag					LR+OS	[44][40]			
C4.5+UOS					SVM+UBst					LR+UOS				
C4.5+SMOTE	[36][40][41]				SVM+OBst					LR+SMOTE	[44][40]			
C4.5+COS	[36][7]		[4][43]		SVM+UOBst					LR+COS				
C4.5+EM1v1	[36]				SVM+SBst					LR+EM1v1				
C4.5+UBag	[42]		[42]		Ripper+Bag	[36]				LR+UBag				
C4.5+OBag					Ripper+Bst	[36]				LR+OBag				
C4.5+UOBag					Ripper+US	[36][40]				LR+UOBag				
C4.5+SBag					Ripper+OS	[36][40]				LR+SBag				
C4.5+UBst					Ripper+UOS					LR+UBst				
C4.5+OBst					Ripper+SMOTE	[36][40]				LR+OBst				
C4.5+UOBst					Ripper+COS	[36]				LR+UOBst				
C4.5+SBst	[7]				Ripper+EM1v1	[36]				LR+SBst				
RF+Bag	[36]				Ripper+UBag					NB+Bag	[36]			
RF+Bst	[36]				Ripper+OBag					NB+Bst	[36]			
RF+US	[36][40]				Ripper+UOBag					NB+US	[36][39][40]			
RF+OS	[36][40]				Ripper+SBag					NB+OS	[36][39][40]			
RF+UOS					Ripper+UBst					NB+UOS				
RF+SMOTE	[36][40]				Ripper+OBst					NB+SMOTE	[36][40]			
RF+COS	[36]				Ripper+UOBst					NB+COS	[36]			
RF+EM1v1	[36]				Ripper+SBst					NB+EM1v1	[36]			
RF+UBag					IBk+Bag					NB+UBag	[42]		[42]	
RF+OBag					IBk+Bst					NB+OBag				
RF+UOBag					IBk+US	[40]				NB+UOBag				
RF+SBag					IBk+OS	[40]				NB+SBag				
RF+UBst					IBk+UOS					NB+UBst				
RF+OBst					IBk+SMOTE	[40]				NB+OBst				
RF+UOBst					IBk+COS					NB+UOBst				
RF+SBst					IBk+EM1v1					NB+SBst				
SVM+Bag					IBk+UBag					-				

Note: Please see Section 4.2 for the interpretation of abbreviations for the defect prediction methods.

TABLE 1: Summary of Previous Experiments on Imbalanced Learners, Classification Methods and Input Metrics for Software Defect Prediction

found that under-sampling beat over-sampling for both Naïve Bayes and C4.5, but Sun et al.'s work indicates this is only true for C4.5.

Second, Seiffert et al. [40] conducted a further study on class imbalance coupled with noise for different classifiers and data sub-sampling techniques. They found that only some classifiers benefitted from the application of sub-sampling techniques in line with Menzies et al. [39] and Sun et al. [36]. However, they also reported conflicts in terms of the performance of random over-sampling methods outperform other sub-sampling methods at different levels of noise and imbalance.

A third example, again from Seiffert et al. [37], is where they compared sub-sampling methods with Boosting for improving the performance of decision tree models built to predict defective components. Their results show that Boosting outperforms even the best sub-sampling methods. In contrast, Khoshgoftaar et al. [43] learned classifiers through using Boosting and cost-sensitive Boosting with C4.5 and decision stumps used as the base classifiers, respectively. They found that Boosting and cost-sensitive Boosting did not enhance the performance of individual pruned C4.5 decision trees.

Therefore, our study focuses on an exhaustive comparison of 16 different popular imbalanced learning methods (plus the control of no Imbalanced Learning) with seven representative and widely used traditional machine learning methods on static code, process, and network metrics in terms of five performance measures in the same experimental context for the purpose of software defect prediction. These are all applied to 27 different data sets.

3 METHOD

Our goal is to conduct a large scale comprehensive experiment to study the *effect* of imbalanced learning and its complex interactions between the type of classifier, data set characteristics and input metrics in order to improve the practice of software defect prediction. We first discuss our choice of MCC as the performance measure and then describe the experimental design including algorithm evaluation, statistical methods and software defect data sets.

3.1 Classification Performance Measures

Since predictive performance is the response variable for our experiments, the choice is important. Although the *F-measure* and *AUC* are widely used, we see them as problematic due to bias particularly in the presence of unbalanced

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

data sets, which is of course precisely the scenario we are interested in studying. Consequently, we use *MCC* (Matthews correlation coefficient [45] (MCC) otherwise known as ϕ - see [46]) as our measure of predictive performance.

	Actually Positive	Actually Negative
Predict Positive	*TP*	*FP*
Predict Negative	*FN*	*TN*

TABLE 2: Confusion Matrix

The starting point for most classification performance measures is the confusion matrix. This represents counts of the four possible outcomes when using a dichotomous classifier to make a prediction (see Table 2)[4]. For example, F_1 is the most commonly used derivative of the *F-measure* family and is defined by Eqn. 1.

$$F_1 = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{1}$$

However, it excludes True Negatives (*TN*) in its calculation which is potentially problematic. The reason is that it originated from the information retrieval domain where typically the number of true negatives, e.g., irrelevant web pages that are correctly not returned is neither knowable nor interesting. However, unlike recommendation tasks[5], this is not so for defect prediction because test managers are definitely interested to know if components are truly non-defective.

Let us compare F_1 with MCC. MCC is the geometric mean of the regression coefficients of the problem and its dual [10] and is defined as:

$$MCC = \frac{\mathrm{TP} \times \mathrm{TN} - \mathrm{FP} \times \mathrm{FN}}{\sqrt{(\mathrm{TP}+\mathrm{FP})(\mathrm{TP}+\mathrm{FN})(\mathrm{TN}+\mathrm{FP})(\mathrm{TN}+\mathrm{FN})}} \tag{2}$$

As a correlation coefficient it measures the relationship between the predicted class and actual class, *MCC* is on a scale [-1,1] where 1 is a perfect positive correlation (also perfect prediction), zero no association and -1 a perfect negative correlation. In contrast, we illustrate the problematic nature of F_1 with a simple example and compare it with *MCC*.

Suppose our hypothetical defect classifier predicts as Case 1 in Table 3.

	Case 1	Case 2	Case 3	Case 4
TP	5	5	4	9
FP	45	45	9	54
FN	5	5	6	1
TN	45	0	81	36
F_1	0.17	0.17	0.35	0.25
MCC	0.00	-0.67	0.27	0.19
G-mean	0.50	0.00	0.60	0.60

TABLE 3: Example Classification Cases

We can see the proportion of cases correctly classified is 0.5 i.e., TP+TN/n = 5+45/100. This yields an F_1 of 0.17 on a scale [0,1] which is somewhat difficult to interpret. Let us compare F_1 with *MCC*. In this case, *MCC*=0 which is intuitively reasonable since there is no association between predicted and actual[6]. Now suppose the True Negatives are removed so n=55 as in Case 2 in Table 3. F_1 remains unchanged at 0.17 whilst *MCC*=-0.67 signifying substantially worse than random performance. The proportion of correctly classified cases is now 5/55 = 0.09, clearly a great deal worse than guessing and so we have a perverse classifier. However, F_1 cannot differentiate between the two situations. This means experimental analysis based upon F_1 would be indifferent to the two outcomes.

This example illustrates not only a drawback with F_1, but also the weakness of all derivative measures from *Recall* and *Precision* as they ignore TNs. Measures such as *Accuracy* are also well-known to be biased as they are sensitive to data distributions and the prevalence of the positive class [49].

One alternative measure that covers the whole confusion matrix is the *G-mean*, defined as the geometric mean of the accuracies of the two classes (see Eqn. 3) and was developed specifically for assessing the performance under imbalanced domains [21]. It assumes equal weight of the precision for both classes.

$$\textit{G-mean} = \sqrt{\frac{TP}{TP+FN} \times \frac{TN}{TN+FP}} \tag{3}$$

However, there are disadvantages with *G-mean*. As observed by López et al.[19], "due to this symmetric nature of the geometric mean ... it is hard to contrast different models according to their precision on each class". For example, in Table 3 we observe that Case 3 ($TP_{rate} = 0.4$, $TN_{rate} = 0.9$) and Case 4 ($TP_{rate} = 0.9$, $TN_{rate} = 0.4$) the G-mean is the same 0.60. However, Case 3 is clearly preferred by MCC and F1. An alternative version called the G-measure replaces TN_{rate} with *precision*, however, it ignores *TN* and suffers the same drawback as the F-measure.

Thus, we seek a single measure that:

1) Covers the entire confusion matrix;
2) Evaluates a specific classifier[7];
3) Properly takes into account the underlying frequencies of true and negative cases;
4) can be easily interpreted

The third requirement needs further discussion in that AUC — another commonly used measure for evaluating classifiers — is also problematic. AUC calculates the area under an ROC curve which depicts relative trade-offs between TPR (true positive rate which is TP/(TP+FN)) and FPR (false positive rate which is FP(FP+TN)) of classification for every possible threshold. One classifier can only be preferred to another if it strictly dominates i.e., every point on the ROC curve of this classifier is above the other curve. Otherwise, we cannot definitively determine which classifier is to be preferred since it will depend upon the relative costs of FPs and FNs.

Consider the example in Fig. 1 that shows ROC curves for two classifiers (Classifier Family A and Classifier Family

4. Note: in the context of software defect prediction, the positive class and negative class denote defective and non-defective respectively.

5. Examples of recommendation tasks include bug triage [47] or recommending code snippets [48].

6. This is a typical random guess where the accuracy for both classes is 50%.

7. As opposed to a family of classifiers such as is the case for the Area Under the Curve (AUC) measure [50]

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

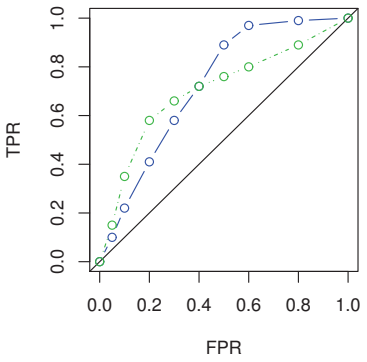

Fig. 1: ROC curves of Classifier A (the solid curve) and Classifier B (the dotdash curve)

FPR	TPR(A)	TPR(B)
0.05	0.1	0.15
0.1	0.22	0.35
0.2	0.41	0.58
0.3	0.58	0.66
0.4	0.72	0.72
0.5	0.89	0.76
0.6	0.97	0.8
0.8	0.99	0.89

AUC	A	B
	0.725	0.704

TABLE 4: Points on the A and B ROC curves

B) derived from the values of some points on these curves (Table 4). We can observe that B is better than A when FPR is less than 0.4, but this reverses when FPR is greater than 0.4. Without knowing the relative costs of FP and FN we cannot determine which classifier is to be preferred. As a compromise, the area under the curve can be calculated to quantify the overall performance of classifier families, i.e. the AUC of A is 0.725 which is greater than the AUC of B (0.704). The AUC values indicate A is better than B, but this still doesn't help us determine which *specific* classifier we should actually choose.

Moreover, AUC is incoherent in that it is calculated on different misclassification cost distributions for different classifiers [51], since various thresholds relate to varying misclassification costs. Hence we conclude AUC is unsuitable for our purposes. Consequently, we select *MCC* as our performance measure. For a fuller discussion of the merits and demerits of various classification performance metrics see [10], [52], [21].

3.2 Algorithm Evaluation

In order to be as comprehensive as possible, we apply a total of 17 different imbalanced learning methods (16 plus a null method - see Table 5) to seven traditional classifiers chosen to be representative of commonly used approaches [1] (see Table 6). We then use seven classes of input metric (see Table 7). Since the design is factorial this yields 833 combinations which are evaluated across 27 different data sets as a repeated measure design as this enables us to compare performance between approaches *within* a given data set.

Then for each combination we use $M \times N$-*way cross-validation* to estimate the performance of each classifier, that is, each data set is first divided into N bins, and after that a predictor is learned on $(N-1)$ bins, and then tested on the remaining bin. This is repeated for the N folds so that each bin is used for training and testing while minimizing the sampling bias. Moreover, each holdout experiment is also repeated M times and in each repetition the folds are randomized. In our case $M = 10$ and $N = 10$ so overall, 100 models are built and 100 results obtained for each data set.

Method Type	Abbr	Method Name	Ref
Sub-sampling methods	US	Under-Sampling	[18]
	OS	Over-Sampling	[18]
	UOS	Underover-Sampling	[18]
	SMOTE	SMOTE	[18]
Cost-sensitive methods	COS	Cost-sensitive learning	[53]
Ensemble methods	Bag	Bagging	[16]
	Bst	Boosting	[17]
Imbalanced ensemble methods	EM1v1	EM1v1	[36]
	UBag	UnderBagging	[14]
	OBag	OverBagging	[33]
	UOBag	UnderoverBagging	[14]
	SBag	SMOTEBagging	[14]
	UBst	UnderBoosting	[34]
	OBst	OverBoosting	
	UOBst	UnderoverBoosting	
	SBst	SMOTEBoosting	[35]

TABLE 5: Summary of imbalanced learning methods

Abbr	Classification Algorithm	Ref
LR	Logistic Regression	[6], [54]
NB	Naïve Bayes	[55]
C4.5	Decision tree	[4]
IBk	Instance based *k*NN	[56]
Ripper	Rule based Ripper	[57]
SVM	Support vector machine (SMO)	[58]
RF	Random Forest	[7]

TABLE 6: Summary of classifiers

The experimental process is shown by the following pseudo-code. Notice that (i) attribute selection is applied to the training data of each base learner, see Lines 14 and 22. (ii) the test data is always 'raw' and imbalanced.

Lastly, our experimental program is based on WEKA [61]. The parameter k=5 for IBk and SMOTE. SVM is the default SMO in WEKA. The number of iterations for all the ensemble methods is 10 except EM1v1 which determines the number itself. COS is the MetaCOS in WEKA with the cost-matrix defined as the same ratio as IR. The remaining parameters are all the default values from WEKA[8].

8. https://www.cs.waikato.ac.nz/ml/weka/index.html

Input Metrics	Metrics Type	Ref
CK	Source Code metrics	[59]
NET	Network metrics	[60]
PROC	Process metrics	[4]
CK+NET	Combined metrics	-
CK+PROC	Combined metrics	-
NET+PROC	Combined metrics	-
CK+NET+PROC	Combined metrics	-

TABLE 7: Input metric classes used in our experiment

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

Procedure Experimental Process

```
1  M ← 10;                                  /*the number of repetitions*/
2  N ← 10;                                  /*the number of folds*/
3  DATA ← {D_1, D_2, ..., D_n};             /*software data sets*/
4  Learners ← {C4.5, RF, SVM, Ripper, IBk, LR, NB};
5  ImbalancedMethods ← {Bag, Bst, US, OS, UOS, SMOTE, COS, EM1v1,
   UBag, UOBag, OBag, SBag, UBst, OBst, UOBst, SBst};
6  for each data ∈ DATA do
7      for each times ∈ [1, M] do           /*M times N-fold
       cross-validation*/
8          data' ← randomize instance-order for data;
9          binData ← generate N bins from data';
10         for each fold ∈ [1, N] do
11             testData ← binData[fold];
12             trainingData ← data' - testData;
13             for each learner ∈ Learners do
                   /*evaluate traditional learning      */
14                 trainingData' ← attributeSelect(trainingData);
15                 classifier ← learner(trainingData');
16                 learnerPerformance ← evaluate classifier on testData;
17             for each imbMethod ∈ ImbalancedMethods do
18                 T ← iteration number of imbMethod;
                   /*build classifiers from each
                   traditional learner                */
19                 for each learner ∈ Learners do
20                     for each t ∈ [1, T] do
21                         D_t ← generateData(t, trainingData,
                           imbMethod);
22                         D'_t ← attributeSelect(D_t);
23                         C_t ← learner(D'_t);
24                     imbClassifier ← ensembleClassifier({C_t,
                       t = 1..T}, imbMethod);
                       /*evaluate imbalanced learning   */
25                     imbPerformance ← evaluate imbClassifier on
                       testData;
```

3.3 Statistical Methods

Given the performance estimates of each classifier on every dataset, how to determine which classifier is better?

First we need to examine whether or not the performance difference between two predictors could be caused by chance. We use a Wilcoxon signed-rank test (a non-parametric statistical hypothesis test used when comparing paired data) to compare pairs of classifiers. Like the sign test, it is based on difference scores, but in addition to analyzing the signs of the differences, it also takes into account the magnitude of the observed differences. The procedure is non-parametric so no assumptions are made about the probability distributions, which is important since a normal distribution is not always guaranteed. We correct for multiple tests by using the Benjamini-Yekutieli step-up procedure to control the false discovery rate [62]. Then the Win/Draw/Loss record is used to summarise each comparison by presenting three values, i.e., the numbers of data sets for which Classifier C_1 obtains better, equal, and worse performance than Classifier C_2.

Next, *effect size* is computed since it emphasises the practical size of the difference, rather than confounding this with sample size as is the case with p-values [63]. We use the effect statistics of difference (average improvement) and dominance (Cliff's δ). Cliff's δ is a non-parametric robust indicator which measures the magnitude of dominance as the difference between two groups [64]. It estimates the likelihood of how often Predictor C_1 is better than Predictor C_2. We use the paired version since our data are correlated so the performance of two methods is comparable for the same data set [65], [66]. By convention, the magnitude of the difference is considered trivial ($|\delta| < 0.147$), small ($0.147 \leq |\delta| < 0.33$), moderate ($0.33 \leq |\delta| < 0.474$), or large ($|\delta| \geq 0.474$) as suggested by Romano et al. [67].

3.4 Software Metrics

As indicated, we are interested in three classes of metric based upon static code analysis, network analysis and process. These choices are made because static code metrics are most frequently used in software defect prediction [68], network metrics may have a stronger association with defects [60] and process metrics reflect the changes to software systems over time. We also consider combinations of these metrics yielding a total of seven possibilities (see Table 7).

(1) Source code metrics measure the 'complexity' of source code and assume that the more complex the source code is, the more likely defects are to appear. The most popular source code metrics suite is the Chidamber-Kemerer (CK) metrics [59] which is detailed in Appendix A.1. All six CK metrics and *LOC* (lines of code) were chosen as code metrics in this paper and marked as CK.

(2) Network metrics are actually social network analysis (SNA) metrics calculated on the dependency graph of a software system. These metrics quantify the topological structure of each node of the dependency graph in a certain sense, and have been found as effective indicators for software defect prediction [60]. In this study, the networks are call graphs of software systems, where the nodes are the components (classes) of a software and the edges are the call dependencies among these components. The DependencyFinder[9] tool was used to extract the call relations. Once networks are built, the UCINET[10] tool was employed to calculate three kinds of network (NET) metrics of dependency networks: Ego network metrics, structural metrics and centrality metrics. The details of 25 types of SNA metrics are given in the Appendix A.2.

(3) Process metrics represent development changes on software projects. We extracted 11 process (PROC) metrics, which were proposed by Moser et al. [4] from the CVS/SVN repository of each specific open source project (see Appendix A.3).

3.5 Data Sets

Public software defect data repositories include PROMISE [69], AEEEM [70], NASA[11] [71], Softlab [72], Relink [73], NetGene [74] and the JiT data repository [75]. In total, we located 106 software defect prediction data sets in these seven repositories. From these data sets we extract imbalance ratios which are described in Section 4.1.

However, we also need to collect data with consistent granularity and input metrics. This requires not only the availability of defect data but also the availability of software metrics, i.e. it is not possible to explore the source code of the NASA MDP data sets and the contextual data are not comprehensive (e.g., no data on maturity are available).

9. http://depfind.sourceforge.net/

10. http://www.analytictech.com/ucinet/

11. Note that although the NASA MDP data sets have been widely used in developing defect prediction models, the data may "suffer from important anomalies." [1].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

Note that we exclude data sets which are too small [12] and too defective[13]. As a result, we are limited to 27 data sets (for our main analysis) from the PROMISE and AEEEM repositories where we are able to collect necessary class-level metrics (CK, NET and PROC). These 27 data sets are derived from 13 distinct software projects, since there are multiple releases for many of these projects (e.g., ant has releases 1.3 to 1.6, see Table 8).

4 Experimental Results and Analysis

In this section, we consider the effect of imbalanced learning meaning a non-zero difference between imbalanced learning and traditional learning, and of course we are most interested in positive effects. We start with the basic characteristics of imbalanced learning problem on software defect prediction by investigating:

- *RQ1* How imbalanced are software defect data sets?
- *RQ2* How does the traditional learning perform under imbalance?
- *RQ3* How does the imbalanced learning compare with traditional learning?

Then we explore more details about the *effect* and its complex interactions with our three experimental factors by answering the following questions expansed from *RQ4*:

- *RQ4.1* How does imbalance level impact performance?
- *RQ4.2* How does the type of classifier interact with imbalance level?
- *RQ4.3* How does type of input metric interact with imbalance level?
- *RQ4.4* How does type of imbalanced learning method interact with imbalance level, type of classifier and type of input metrics?

4.1 RQ1: Imbalance Levels in Defect Data Sets

As previously mentioned in Section 3.5, we identified a total of 106 public software defect prediction data sets that at least provide sufficient information for us to compute the Imbalance Ratios (see Fig 2). Since the distribution of *IR* is highly skewed (Skewness =7.16, Kurtosis = 54.84) we plot the histogram of $log_2(IR)$. The blue bars indicate the 27 data sets selected for the remainder of the analysis. The dotted line shows the median of all data sets ($2^{1.98} = 3.94$).

As we can observe from Fig 2, most data sets gather around the median in the $log_2(IR)$ range from -1 to 4. It is customary to categorize imbalance in terms of *IR* orders of magnitude [88], [29]. We find there are only 13 out of 106 data sets whose *IR* is larger than 10. In other words, the imbalance level of 87.7% software defect data is not as extreme as in some other domains such as fraud detection or network intrusion [29]. This is also a pointer for our study in that low and medium levels of imbalance are more typical. Thus, the 27 data sets are divided into two groups: (i) Low IR (ii) Moderate+ (moderate and high) IR by splitting on the median ($IR = 3.94$). The labels are chosen to align with the wider context of data imbalance [29].

12. 25 data sets whose size is less than 120 components are filtered out.

13. We assume the defective class is in the minority (i.e., $IR > 1$).

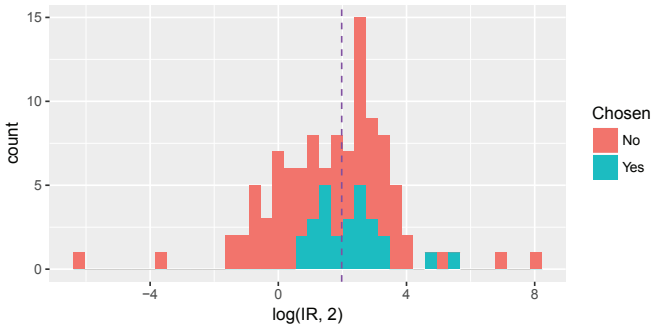

Fig. 2: Histogram of Data set Imbalance Ratios ($log_2(IR)$)

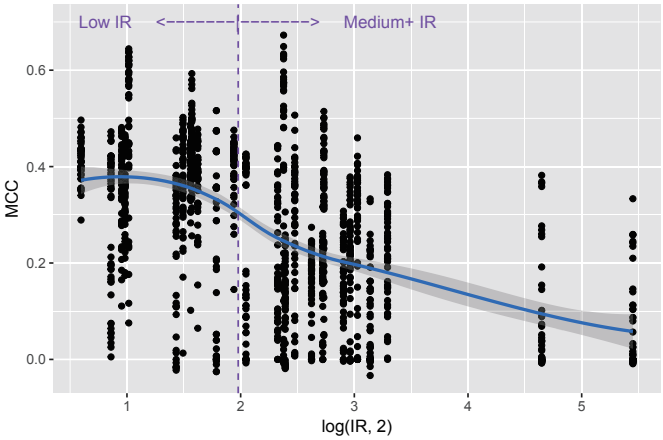

Fig. 3: Performance without imbalanced learning under differing levels of data imbalance

Another observation is that fortunately the distribution of blue bars is similar to the whole distribution, indicating our subset is reasonably representative of the whole distribution. Recall that our choice of data sets is to include all data sets where are able to compute each class of input metric, there are at least 120 components and the defective class is the minority, i.e., $IR > 1$ since we take the view that if defective components form the majority, then it implies the system is quite problematic perhaps being an initial version of a software project.

In summary, a typical level of imbalance is a ratio of approximately 4 and 87.7% of software defect data sets are not highly imbalanced, i.e., low and moderate levels of imbalance are more prevalent.

4.2 RQ2 How does traditional learning perform under imbalance?

To answer this question, we plot the predictive performance (MCC) without imbalanced learning under different levels of imbalance in Fig. 3. The blue line is drawn by a non-parameter smoother (loess smoothing) with its confidence interval shown as a gray area. As per Fig 2, we plot $log_2(IR)$ instead of IR. Again the dotted line shows the median ($2^{1.98} = 3.94$).

From this enhanced scatter plot, we see a great deal of variability in predictive performance for any given level of imbalance but also observe that $log_2(IR)$ tends to have a negative impact upon classification performance. The

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

Dataset	Modules	Defect Ratio	Imbalance Ratio	LOC Avg	LOC Max	Revision Avg	Reference
ant-1.3	125	16.00%	5.25	301.59	2193.00	2.90	[76], [77]
ant-1.4	178	22.47%	3.45	304.47	2040.00	2.57	[76], [77]
ant-1.5	293	10.92%	8.16	297.09	4244.00	7.71	[76], [77], [78]
ant-1.6	351	26.21%	2.82	322.64	4238.00	8.32	[76], [77], [78]
camel-1.0	339	3.83%	25.08	99.47	1000.00	4.69	[76], [77]
camel-1.2	608	35.53%	1.81	109.05	1056.00	1.04	[76], [77], [78]
camel-1.4	872	16.63%	5.01	112.48	1747.00	1.55	[76], [79], [77], [78]
camel-1.6	965	19.48%	4.13	117.16	2077.00	0.44	[76], [79], [77], [80], [81], [78]
ivy-2.0	352	11.36%	7.80	249.34	2894.00	8.00	[76], [82], [77], [78]
jedit-3.2	272	33.09%	2.02	473.83	23350.00	1.09	[76], [77], [83] , [84], [78]
jedit-4.0	306	24.51%	3.08	473.21	23683.00	4.63	[85], [76], [77], [83] , [84], [80], [78]
jedit-4.1	312	25.32%	2.95	490.66	23590.00	4.16	[85], [83], [84], [78]
jedit-4.2	367	13.08%	6.65	465.08	12200.00	6.13	[85], [83], [84]
jedit-4.3	492	2.24%	43.73	411.31	12535.00	7.30	[85], [82], [83], [84]
log4j-1.0	135	25.19%	2.97	159.62	1176.00	2.38	[80], [81], [78]
poi-2.0	314	11.78%	7.49	296.72	9849.00	3.86	[76], [77]
synapse-1.0	157	10.19%	8.81	183.48	867.00	5.20	[76], [77], [78]
synapse-1.1	222	27.03%	2.70	190.55	1164.00	3.22	[76], [77], [78]
synapse-1.2	256	33.59%	1.98	208.98	1449.00	3.27	[76], [82], [77], [78]
velocity-1.6	229	34.06%	1.94	248.96	13175.00	2.40	[76], [82], [77]
xerces-1.2	440	16.14%	5.20	361.94	8696.00	1.21	[76], [77], [86], [78]
xerces-1.3	453	15.23%	5.57	368.86	10701.00	1.00	[76], [77], [86], [80], [81],[78]
Eclipse_JDT_Core	997	20.66%	3.84	224.73	7341.00	45.62	[70], [87], [86]
Eclipse_PDE_UI	1497	13.96%	6.16	98.16	1326.00	13.51	[70], [87],[81]
Equinox_Framework	324	39.81%	1.51	122.02	1805.00	11.39	[70], [87],[81]
Lucene	691	9.26%	9.80	105.91	2864.00	6.26	[70], [76], [87] , [79] , [77], [82] , [86], [80], [81]
Mylyn	1862	13.16%	6.60	83.84	7509.00	10.98	[70], [81]

TABLE 8: Description of the 27 Data Sets

smoothed line starts at nearly MCC=0.4, decreases to 0.3 at the median (the dashed line) then further falls to below 0.1 MCC where $log_2(IR)$ exceed 8 i.e, approximately 64. In other words at this level of imbalance the performance of classifiers is scarcely better than random. More generally, the negative impact is clear upon the right side of dotted line for the Medium+ IR data sets (defined in Section 4.1) where this imbalance can potentially cause a substantial reduction in predictive performance.

In addition, the robust correlation coefficient[14] is -0.529 ($p < 0.0001$). This indicates a negative correlation between the performance and $log_2(IR)$ which would generally be regarded as a large effect [90].

In summary, the answer for *RQ2* is that the performance of traditional learning is highly threatened by moderate or high levels of imbalance in software defect data. Therefore any means of addressing imbalance in the data is potentially useful for software defect prediction.

4.3 RQ3: How does imbalanced learning compare with traditional learning?

To answer our next research question *RQ3*, we compare traditional learning methods and imbalanced learning methods pairwise (for the same base classifier, metrics and data set) and then analyze the overall differences. This follows from the repeated measure design of the experiment described in Section 3.2. From this we can compute the *difference* between predicting defects with and without an imbalanced learner for each data set.

Fig. 4 shows the distribution of the *effect* (differences) as a histogram. The shaded bars in the histogram indicate negative effects, i.e., where the imbalanced learning makes the predictive performance worse. Overall, this happens in about 28% of the cases. There are several factors which may cause such negative effect:

1) low level imbalance;
2) the base classifier is not sensitive to imbalance;
3) a weak imbalanced learning method;
4) challenging datasets e.g., problems with class separability and within-class sub-concepts [29]

For first three factors, we offer evidence and analysis in Section 4.4, Section 4.5 and Section 4.7 respectively.

Yet careful examination of these negative cases suggests that imbalanced learning can be also counter-productive, due to the lack of structure to learn from for challenging datasets (rare for strong learners). Interestingly the fourth factor is not limited to software defects. Lopez et al. [19] identify a number of intrinsic data set characteristics that have a strong influence on imbalanced classification, namely, small disjuncts, lack of density, lack of class separability, noisy data and data set shift. They also pointed out that poor performance can occur across the range of imbalance ratios.

This means imbalanced learning is not a simple panacea for all situations. It must be carefully chosen and applied

14. We use the percentage bend correlation coefficient [89] from pbcor in the WRS2 R package.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

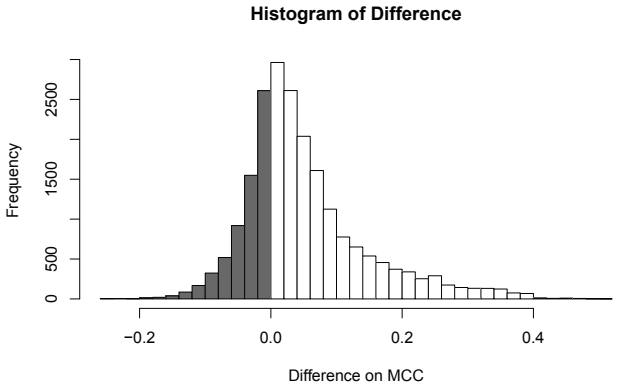

Fig. 4: Histogram of differences in predictive performance (MCC) with and without imbalanced learning

on software defect prediction. In the following sections we report Cliff's δ and drill deeper to better understand factors that are conducive to successful use of imbalanced learning.

Statistic	Value	Statistic	Value
Min	-0.258	Max	0.504
Mean	0.051	Trimmed mean	0.035
sd	0.093	Trimmed (0.2) sd	0.071
Skewness	1.207	Kurtosis	1.926

TABLE 9: Summary statistics for differences in performance (MCC) with and without imbalanced learning

Table 9 provides summary statistics of the differences. We see that the standard deviation (sd) and trimmed sd are both larger than mean or trimmed mean. This indicates that there is a good deal of variability in impact and that the possibility of negative effects from imbalanced learning cannot be ignored. The kurtosis equals to 1.926 which implies a considerably fat-tailed distribution. This all indicates that our analysis needs robust statistics. Therefore using the percentile bootstrap method [89] we can estimate the 95% confidence limits around the trimmed mean 0.035 as being (0.034, 0.036). We can be confident the typical difference between using an imbalanced learner or not is a small positive effect on the ability to predict defect prone software. So the answer for *RQ3* is that overall the *effect* is small but positive, however, this is in the context that there are about 28% negative results.

4.4 RQ4.1: How does imbalance level impact performance?

It is possible that imbalanced learning methods perform less well on the data sets with low level of imbalance. Thus we investigate *RQ4.1* in two groups of imbalance described in Section 4.1: (i) Low IR (ii) Medium+ IR as shown in Fig. 5. The notches show the 95% confidence intervals around the median (shown as a thick bar) as same as the other boxplots.

We observe the *effect* (dMCC as difference in MCC) on Medium+ IR group is greater than the *effect* on the Low IR group. This is to be expected since traditional learning is harmed by moderate or higher imbalance. But

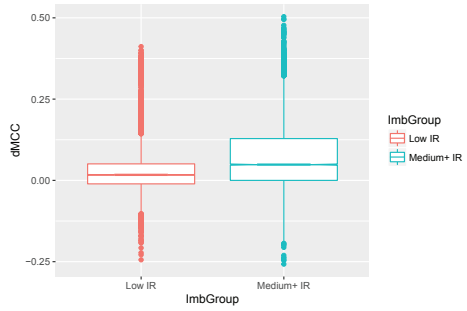

Fig. 5: Boxplot of differences in performance (MCC) with and without imbalanced learning by IR groups

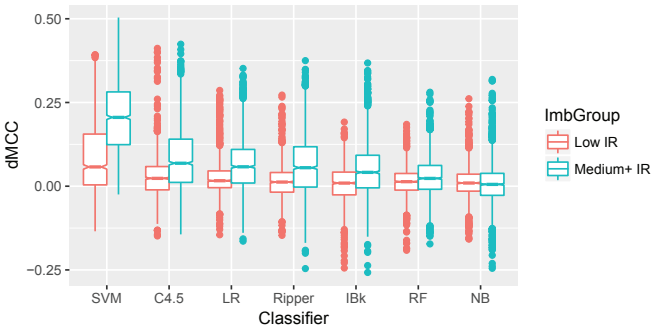

Fig. 6: Boxplot of differences in performance (MCC) with and without imbalanced learning by classifier type

again it also offers a warning that positive effects are not guaranteed since the boxplot whiskers go below zero for both two groups. Thus we present the *effect size* as both average improvement (difference in MCC) and dominance statistics (Cliff's δ) which show the stochastic likelihood that imbalanced learning is better than nothing.

Level	Average Improvement	Cliff's δ
Low IR	0.019 (0.018, 0.02)	0.316 (0.297, 0.335)
Medium+ IR	0.055 (0.054, 0.057)	0.501 (0.485, 0.517)

TABLE 10: Effect Size with CI by Imbalance Levels

As shown in Table 11, for Low IR group both the average improvement (0.019) and Cliff's δ is small ($\delta < 0.33$). In the contrast, the average improvement for Medium+ IR group is 0.055 and Cliff's δ is large ($\delta > 0.474$), which indicates a strong likelihood of obtaining a positive effect. This is to be expected since traditional learning is harmed by moderate or higher imbalance as shown in Section 4.2. Therefore the answer for *RQ4.1* is that the *effect* for Low IR group is small but for Medium+ group is large.

4.5 RQ4.2: How does the type of classifier interact with imbalance level?

In our experiment *RQ4.2* we investigate seven different types of classifier (listed in Table 6). Fig. 6 shows the difference i.e., the *effect* achieved by introducing an imbalanced learning algorithm as boxplots organised by classifier types. The red boxes and blue boxes indicate Low IR group and Medium+ IR group, respectively. In this way, we can

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

visualize not only the effect of classifier type, but also its interaction with IR groups. We also observe Medium+ IR impacts the predictive difference of performance such that the benefits of using an imbalanced learner are greater when the classifier is sensitive to IR. Also there is more variance with Medium+ imbalance as indicated by the larger boxes.

However the impact of imbalance is relatively limited other than for SVM. Support vector machines (SVM) consistently benefit from imbalanced learning. This is in line with Batuwita and Palade [91] who report that the "separating hyperplane of an SVM model developed with an imbalanced dataset can be skewed towards the minority class, and this skewness can degrade the performance of that model with respect to the minority class". In all cases, there are long whiskers suggesting high variability of performance and in all cases the whiskers extend below zero suggesting the possibility (though falling outside the 95% confidence limits given in Table 11) of a deleterious or negative effect. Therefore we provide both the average improvement and the probability of obtaining an improvement through an imbalanced learner.

Classifier	Average Improvement	Cliff's δ
Medium+ IR group		
SVM	0.204 (0.198, 0.210)	0.893 (0.872, 0.912)
C4.5	0.072 (0.067, 0.077)	0.593 (0.553, 0.630)
LR	0.058 (0.054, 0.062)	0.624 (0.585, 0.660)
Ripper	0.056 (0.052, 0.061)	0.475 (0.432, 0.516)
IBk	0.043 (0.039, 0.046)	0.457 (0.414, 0.499)
RF	0.024 (0.022, 0.027)	0.365 (0.320, 0.409)
NB	0.005 (0.003, 0.008)	0.101 (0.053, 0.148)

TABLE 11: Effect Size with CI by Type of Classifier

Table 11 presents the *effect size* of both statistics for each type of classifier on Medium+ IR group (in the decreasing order of the average improvement), since the impact of Low IR is limited. We observe considerable spread from a maximum average improvement of 0.204 to a negligible 0.005, likewise with Cliff's δ varying from a very large (0.893) to a trivial (0.101). As we can see the large positive effect for SVM confirms that there are opportunities to improve SVM by imbalanced learning. The effect size also reflects the sensitivity of each type of the classifiers to the IR. Naïve Bayes is insensitive to an imbalanced distribution with an average improvement of less than 0.01.

So the answer for *RQ4.2* is classifier sensitivity to imbalance is most noticeable for the Medium+ imbalance group.

4.6 RQ4.3: How does type of input metric interact with imbalance level?

Next in *RQ4.3* we consider the role of input metric type. The seven different classes are summarized in Table 7 and the distributions of the difference on predictive performance through imbalanced learning shown as boxplots grouped by Metric in Fig. 7. Again the red boxes and blue boxes indicate Low IR group and Medium+ IR group, respectively. Overall we see much similarity between the boxplots suggesting relatively little difference between Metric type. Also we see

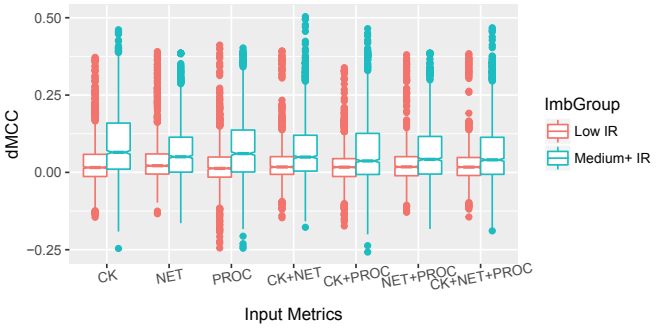

Fig. 7: Boxplot of differences in performance (MCC) with and without imbalanced learning by metric type

that for Low IR very little impact, i.e., the effect sizes are close to zero contrasting with Medium+ IR. Thus we focus on the *effect size* on Medium+ IR group as shown in Table 12.

Input Metrics	Average Improvement	Cliff's δ
Medium+ IR group		
CK	0.075 (0.069, 0.081)	0.636 (0.598, 0.671)
NET	0.053 (0.049, 0.057)	0.512 (0.470, 0.552)
PROC	0.064 (0.058, 0.069)	0.521 (0.479, 0.560)
CK+NET	0.056 (0.052, 0.061)	0.547 (0.506, 0.586)
CK+PROC	0.048 (0.043, 0.053)	0.411 (0.367, 0.454)
NET+PROC	0.048 (0.043, 0.053)	0.443 (0.399, 0.485)
CK+NET+PROC	0.046 (0.041, 0.051)	0.438 (0.394, 0.479)

TABLE 12: Effect Size with CI by Input Metrics

From Table 12 it can be observed that the effect size of each type of input metric is not significantly different. The range of both average improvement ([0.046, 0.075]) and Cliff's δ ([0.438, 0.636]) is narrower compared with the factor of classifier type. This indicates limited difference in responsiveness to imbalanced learning by type of input metric [15].

4.7 RQ4.4: How does type of imbalanced learning method interact with imbalance level, type of classifier and type of input metrics?

The final research question *RQ4.4*, investigates the impact by specific imbalanced learner. Fig. 8 shows red and blue boxplots for the difference each algorithm makes over no algorithm. Once again we find the same pattern that blue boxes (i.e., Medium+ imbalance) show greater prediction improvement except for Bag. This is unsurprising, because Bag does not balance training data in sampling; instead it aims to approximate a real target from different directions and is not inherently designed to deal with imbalanced data. SMOTE performed better probably because it brings novel information by interpolating between existing ones data points.

In addition, it can be seen that all types of imbalanced methods are capable of producing negative impacts upon

15. Here there is evidence of much more of an effect between the different input metrics with the best performance from the widest range of input metrics (CK+NET+PROC). However it still has limited impact on the difference (*effect*)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

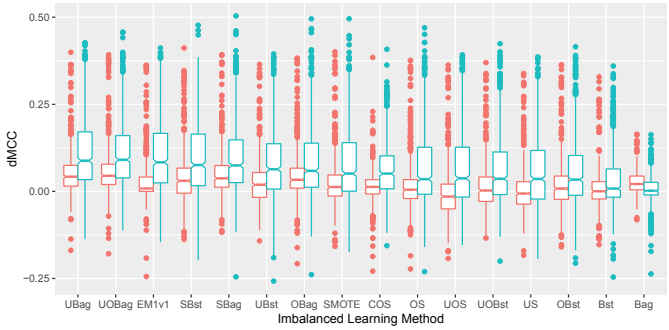

Fig. 8: Boxplot of differences in performance (MCC) with and without imbalanced learning by imbalanced learner

the predictive capability of a classifier. In such case, dominance statistics are useful to quantify the likelihood that one is better than another (see Table 13).

Table 13 shows the average improvement on Cliff's δ for Medium+ IR group, since the impact of Low IR is limited. The values in parentheses give the lower and upper bounds of 95% confidence limits. Note that the algorithms are organised in decreasing order of improvement ranging from 0.096 to 0.005. This spread of improvement (and also for Cliff's δ from 0.732 to 0.144) indicates that the choice of imbalanced methods is very important. Strong imbalanced methods include UBag, UOBag, EM1v1, SBst and SBag.

ImbMethod	Improvement	Cliff's δ
Medium+ IR group		
UBag	0.096 (0.089, 0.105)	0.732 (0.679, 0.777)
UOBag	0.095 (0.088, 0.102)	0.782 (0.733, 0.823)
EM1v1	0.089 (0.081, 0.097)	0.693 (0.637, 0.741)
SBst	0.084 (0.076, 0.093)	0.646 (0.588, 0.698)
SBag	0.079 (0.072, 0.086)	0.721 (0.667, 0.767)
UBst	0.068 (0.061, 0.076)	0.537 (0.474, 0.596)
OBag	0.066 (0.059, 0.074)	0.597 (0.536, 0.652)
SMOTE	0.060 (0.052, 0.068)	0.512 (0.447, 0.571)
COS	0.052 (0.047, 0.058)	0.596 (0.536, 0.65)
OS	0.045 (0.038, 0.053)	0.399 (0.33, 0.463)
UOS	0.045 (0.038, 0.054)	0.347 (0.277, 0.413)
UOBst	0.043 (0.036, 0.050)	0.407 (0.339, 0.471)
US	0.043 (0.035, 0.051)	0.336 (0.266, 0.402)
OBst	0.038 (0.031, 0.045)	0.385 (0.316, 0.45)
Bst	0.015 (0.011, 0.020)	0.185 (0.113, 0.255)
Bag	0.005 (0.003, 0.007)	0.144 (0.074, 0.213)

TABLE 13: Effect Size with CI by Imbalanced Method

To inspect the interactions between type of imbalanced methods, type of classifiers and type of input metrics, Table 14 summarises the results broken down by these three factors as win/draw/loss counts from the 27 data sets in our experiment. We then use the Benjamini-Yekutieli step-up procedure [62] to determine significance[16]. This is indicated by the graying out the *non-significant* cells. The Table is organized to show the Win/Draw/Loss (*W/D/L*) records of comparisons between imbalanced learners in the first row and traditional learners in the first column over the seven different types of metric data shown in the second column. Imbalanced methods and types of classifier are sorted in the same order as 11 and Tables 13. This means the 'better' choices for these two factors are located to the upper left of this table.

So in Table 14 we focus on the white areas as these represent statistically significant results of prediction improvement and show the intersection of Imbalanced Learning algorithm, base classifier and type of metric. From this we derive three major findings.

1) There is greater variability in the performance of the imbalanced learning algorithms compared with traditional learners. Yet again this reveals that not all imbalanced learning algorithms can improve the performance of every classifier and indeed no algorithm is always statistically significantly better.
2) The majority of the unshaded cells are located on the upper left quadrant of this table. This is confirms the importance of using strong imbalanced methods and sensitive classifiers. It also indicates that if the imbalanced learning algorithm, classifier and input metrics can be carefully chosen, there are good opportunities to improve predictive performance.
3) The cells, that show the opposite tendency "white to upper left, shaded to lower right", can still show unexpected interactions between the three factors. For example, all 16 learner types, excluding COS, *can* improve SVM for all input metrics. So there is an interaction between SVM and COS which causes a special case for SVM. This means on the one hand the effect of unexpected interactions is less important than each factor, on the other hand the border choices should be always cautious.

As Table 14 illustrates, we can answer *RQ4.4* by noting the choice of imbalanced methods is important: strong imbalanced methods and sensitive classifiers are preferred.

5 Threats to Validity

In this section, we identify factors that may threaten the validity of our results and present the actions we have taken to mitigate the risk.

The first possible source of bias is the data used. To alleviate this potential threat, 27 defect data sets from 13 public domain software projects were selected from our exhaustive set of 106 publicly available data sets. The remaining 79 were excluded because they provide insufficient metrics for analysis or were very small ($<$ 120 cases) or had exceptionally high fault rates. These data sets are characterized by (i) they are drawn from different types of

16. We need a correction procedure such as Benjamini-Yekutieli since we are carrying out a large (784 to be exact) number of significance tests. We prefer this more modern approach based on a false discovery rate than other conservative corrections such as Bonferroni.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

	Metrics	UBag	UOBag	EM1v1	SBst	SBag	UBst	OBag	SMOTE	COS	OS	UOS	UOBst	US	OBst	Bst	Bag
SVM	CK	25/0/2	25/0/2	21/3/3	25/0/2	25/0/2	24/0/3	25/0/2	25/0/2	9/7/11	25/0/2	25/0/2	25/0/2	25/0/2	25/0/2	20/2/5	14/11/2
	NET	25/1/1	25/1/1	23/1/3	24/1/2	25/1/1	24/1/2	25/1/1	24/1/2	16/2/9	25/1/1	23/0/4	24/0/3	23/1/3	25/0/2	22/0/5	17/4/6
	PROC	27/0/0	26/0/1	26/0/1	25/0/2	25/0/2	26/0/1	25/0/2	25/0/2	7/7/13	25/0/2	27/0/0	26/0/1	27/0/0	27/0/0	20/4/3	11/9/7
	CK+NET	27/0/0	26/0/1	25/1/1	26/0/1	26/0/1	26/0/1	26/0/1	26/0/1	15/0/12	26/0/1	23/0/4	25/0/2	25/0/2	24/0/3	23/0/4	20/1/6
	CK+PROC	26/0/1	26/0/1	22/1/4	26/0/1	26/0/1	25/0/2	24/0/3	26/0/1	10/2/15	24/0/3	26/0/1	26/0/1	25/0/2	25/0/2	24/0/3	17/5/5
	NET+PROC	25/1/1	25/1/1	23/1/3	23/1/3	25/1/1	24/1/2	25/1/1	25/0/2	14/1/12	25/1/1	24/0/3	25/0/2	24/0/3	25/0/2	24/0/3	21/2/4
	CK+NET+PROC	24/1/2	25/1/1	24/1/2	23/1/3	25/0/2	23/1/3	25/1/1	25/0/2	17/1/9	25/0/2	24/0/3	25/0/2	23/0/4	25/0/2	23/0/4	22/2/3
C4.5	CK	24/0/3	24/0/3	18/4/5	21/0/6	25/0/2	18/0/9	21/0/6	18/0/9	22/0/5	18/0/9	17/0/10	17/0/10	17/0/10	16/0/11	16/0/11	17/1/9
	NET	26/0/1	27/0/0	20/4/3	24/0/3	26/0/1	21/0/6	25/0/2	24/0/3	26/0/1	17/0/10	16/0/11	18/0/9	18/0/9	21/0/6	19/0/8	21/0/6
	PROC	23/0/4	21/0/6	18/4/5	21/0/6	22/1/4	19/0/8	21/0/6	19/0/8	24/0/3	15/0/12	16/0/11	16/0/11	19/0/8	16/0/11	15/0/12	17/0/10
	CK+NET	27/0/0	27/0/0	21/2/4	27/0/0	27/0/0	19/0/8	27/0/0	21/0/6	27/0/0	19/0/8	15/0/12	22/0/5	16/0/11	22/0/5	22/0/5	23/0/4
	CK+PROC	21/0/6	22/0/5	19/4/4	21/0/6	21/0/6	16/0/11	20/0/7	17/0/10	22/0/5	11/0/16	11/0/16	15/0/12	15/0/12	17/0/10	16/0/11	19/0/8
	NET+PROC	25/0/2	26/0/1	18/4/5	26/0/1	24/0/3	19/0/8	22/0/5	16/0/11	21/0/6	11/0/16	8/0/19	19/0/8	14/0/13	19/0/8	20/0/7	23/0/4
	CK+NET+PROC	27/0/0	27/0/0	22/3/2	25/0/2	25/0/2	16/0/11	23/0/4	14/0/13	25/0/2	12/0/15	9/0/18	19/0/8	12/0/15	19/0/8	19/0/8	23/0/4
LR	CK	26/0/1	24/0/3	20/4/3	20/0/7	24/0/3	25/0/2	25/0/2	22/0/5	23/0/4	23/0/4	23/0/4	23/0/4	23/0/4	22/0/5	13/0/14	17/1/9
	NET	22/0/5	23/0/4	21/4/2	20/0/7	23/1/3	22/0/5	24/0/3	18/0/9	21/0/6	18/0/9	16/0/11	17/0/10	16/0/11	17/0/10	13/0/14	18/1/8
	PROC	23/0/4	24/0/3	19/4/4	19/0/8	24/0/3	23/0/4	21/1/5	20/0/7	21/0/6	22/0/5	20/0/7	21/0/6	21/0/6	22/0/5	15/0/12	13/0/14
	CK+NET	26/0/1	27/0/0	22/4/1	20/0/7	25/0/2	23/0/4	27/0/0	20/0/7	23/0/4	24/0/3	16/0/11	16/0/11	17/0/10	18/0/9	14/0/13	19/0/8
	CK+PROC	23/0/4	23/0/4	20/4/3	18/0/9	24/0/3	21/0/6	24/0/3	20/0/7	22/0/5	21/0/6	20/0/7	19/0/8	17/0/10	21/0/6	13/0/14	18/0/9
	NET+PROC	24/0/3	23/0/4	19/4/4	21/0/6	23/0/4	20/0/7	21/0/6	18/0/9	23/0/4	19/0/8	14/0/13	16/0/11	15/0/12	16/0/11	15/0/12	22/0/5
	CK+NET+PROC	24/0/3	23/0/4	18/4/5	20/0/7	23/0/4	22/0/5	23/0/4	20/0/7	24/0/3	20/0/7	13/0/14	14/0/13	12/0/15	18/0/9	15/0/12	20/0/7
Ripper	CK	20/0/7	22/0/5	17/0/10	20/0/7	24/1/2	19/0/8	21/0/6	17/0/10	23/0/4	17/0/10	15/0/12	16/0/11	16/0/11	13/0/14	10/0/17	13/0/14
	NET	24/0/3	27/0/0	22/0/5	21/0/6	26/0/1	17/0/10	26/0/1	20/0/7	22/0/5	15/0/12	14/0/13	17/0/10	15/0/12	13/0/14	11/0/16	16/1/10
	PROC	21/0/6	24/0/3	20/0/7	23/0/4	22/0/5	20/0/7	20/0/7	18/0/9	23/0/4	14/0/13	14/0/13	17/0/10	16/0/11	11/0/16	10/0/17	12/0/15
	CK+NET	23/0/4	26/0/1	20/0/7	22/0/5	25/0/2	15/0/12	26/0/1	14/0/13	23/0/4	14/0/13	11/0/16	15/0/12	13/0/14	20/0/7	14/0/13	17/0/10
	CK+PROC	22/0/5	23/0/4	20/0/7	20/0/7	22/0/5	16/0/11	22/0/5	16/0/11		13/0/14	12/0/15	11/0/16	15/0/12	11/0/16	10/0/17	17/0/10
	NET+PROC	23/0/4	25/0/2	23/0/4	24/0/3	24/0/3	15/0/12	24/0/3	19/0/8	22/0/5	11/0/16	10/0/17	15/0/12	14/0/13	18/0/9	14/0/13	17/0/10
	CK+NET+PROC	25/0/2	27/0/0	22/0/5	24/0/3	26/0/1	15/0/12	25/0/2	16/0/11	21/0/6	13/0/14	8/0/19	16/0/11	12/0/15	17/0/10	16/0/11	21/0/6
IBk	CK	22/0/5	22/0/5	19/4/4	20/0/7	22/1/4	22/0/5	18/0/9	19/0/8	17/1/9	15/0/12	9/0/18	14/0/13	13/0/14	12/0/15	8/0/19	15/0/12
	NET	22/0/5	24/0/3	21/4/2	21/0/6	25/0/2	22/0/5	23/0/4	19/0/8	21/0/6	18/0/9	7/0/20	19/0/8	13/0/14	19/0/8	17/0/10	21/0/6
	PROC	25/0/2	24/0/3	20/4/3	24/0/3	21/0/6	23/0/4	20/0/7	21/0/6	23/0/4	18/0/9	15/0/12	16/0/11	18/0/9	15/0/12	9/0/18	12/0/15
	CK+NET	21/0/6	23/0/4	17/4/6	22/0/5	25/0/2	19/0/8	21/0/6	17/0/10	20/0/7	13/0/14	8/0/19	14/0/13	10/0/17	14/0/13	14/0/13	20/0/7
	CK+PROC	23/0/4	22/0/5	20/4/3	17/0/10	23/0/4	20/0/7	16/0/11	15/0/12	18/0/9	11/0/16	7/0/20	13/0/14	10/0/17	8/0/19	5/0/22	17/0/10
	NET+PROC	22/0/5	24/0/3	21/4/2	21/0/6	27/0/0	21/0/6	22/0/5	19/0/8	22/0/5	14/0/13	6/0/21	15/0/12	12/0/15	14/0/13	16/0/11	18/0/9
	CK+NET+PROC	22/0/5	25/0/2	19/4/4	19/0/8	27/0/0	19/0/8	20/0/7	15/0/12	20/0/7	11/0/16	6/0/21	15/0/12	10/0/17	13/0/14	15/0/12	18/0/9
RF	CK	25/0/2	25/0/2	21/0/6	24/0/3	23/0/4	19/0/8	20/0/7	18/0/9	22/0/5	16/0/11	13/0/14	15/0/12	12/0/15	16/0/11	13/0/14	19/0/8
	NET	24/1/2	26/0/1	21/0/6	17/0/10	22/0/5	14/0/13	21/0/6	21/0/6	24/0/3	17/0/10	15/0/12	12/0/15	9/0/18	16/0/11	17/0/10	17/0/10
	PROC	24/0/3	24/0/3	21/1/5	19/0/8	20/0/7	18/0/9	19/0/8	20/0/7	20/0/7	17/0/10	16/0/11	14/1/12	17/0/10	14/0/13	12/0/15	13/0/14
	CK+NET	25/0/2	26/0/1	22/0/5	20/0/7	21/0/6	13/0/14	22/0/5	19/0/8	22/0/5	18/0/9	14/0/13	13/0/14	5/0/22	18/0/9	17/0/10	16/0/11
	CK+PROC	26/0/1	26/0/1	20/0/7	23/0/4	24/0/3	16/0/11	25/0/2	19/0/8	23/0/4	12/0/15	11/0/16	12/0/15	7/0/20	19/0/8	17/0/10	21/0/6
	NET+PROC	24/0/3	25/0/2	18/0/9	20/0/7	22/0/5	11/0/16	19/0/8	19/0/8	22/0/5	13/0/14	11/0/16	13/0/14	8/0/19	17/0/10	16/0/11	20/0/7
	CK+NET+PROC	24/0/3	26/0/1	19/0/8	20/0/7	21/0/6	9/0/18	19/0/8	16/1/10	21/0/6	15/0/12	10/0/17	15/0/12	5/0/22	22/0/5	21/0/6	18/0/9
NB	CK	24/0/3	23/0/4	20/4/3	22/0/5	22/0/5	23/0/4	23/0/4	19/1/7	21/0/6	24/0/3	23/0/4	22/0/5	22/0/5	21/0/6	16/0/11	22/0/5
	NET	20/0/7	18/0/9	15/4/8	16/0/11	19/0/8	18/0/9	17/0/10	13/0/14	17/0/10	16/0/11	14/0/13	10/0/17	11/0/16	10/0/17	7/0/20	17/0/10
	PROC	15/0/12	16/0/11	14/4/9	12/0/15	13/0/14	15/0/12	12/0/15	12/0/15	16/0/11	13/0/14	10/0/17	9/0/18	11/0/16	7/0/20	8/0/19	14/0/13
	CK+NET	21/0/6	20/0/7	15/4/8	14/0/13	19/0/8	19/0/8	18/0/9	13/0/14	15/0/12	18/0/9	17/0/10	15/0/12	14/0/13	13/0/14	11/0/16	20/0/7
	CK+PROC	19/0/8	19/0/8	16/4/7	17/0/10	18/0/9	18/0/9	15/0/12	15/0/12	15/0/12	15/0/12	14/0/13	12/0/15	10/0/17	12/0/15	10/0/17	17/0/10
	NET+PROC	18/0/9	18/0/9	15/4/8	14/0/13	16/0/11	15/0/12	18/0/9	14/0/13	13/0/14	15/0/12	11/0/16	8/0/19	9/0/18	9/0/18	8/0/19	18/1/8
	CK+NET+PROC	16/0/11	17/0/10	14/4/9	15/0/12	19/0/8	13/0/14	18/0/9	14/0/13	11/0/16	13/0/14	12/0/15	11/0/16	10/0/17	11/0/16	7/0/20	18/0/9

Note:
(1) Each cell contains the W/D/L counts of each dataset so $W + D + L = 27$, where W/D/L means the number of data sets for which the classifier in first row of Table 14 obtain better, equal, and worse performance than the classifier in first column of Table 14.
(2) Shaded table cells imply a p-value greater than the threshold suggested by FDR-control procedure, i.e., the predictive performance with imbalanced learning is not significantly better than the performance without imbalanced learning.

TABLE 14: *W/D/L* records of the comparison between predictive performance with and without imbalanced learning

publicly available software projects, (ii) the data sets range considerably in size from 125 to 2196 modules and (iii) there are wide variations in the imbalance ratio between 1.51 and 43.73. These data sets are also widely used by many other researchers. This enables comparison with findings from other experiments. Moreover, we have made our raw results and scripts available (a link is provided at the end of this paper). Nevertheless, to extend our findings it would be valuable to investigate software defect data not drawn from the open software community.

Stochastic data processing techniques, such as sampling or dividing data into training sets and testing sets, also can threaten validity. For this reason we have used 10×10-fold cross-validation in order to mitigate the effects of variability due to the random allocation. This is a well-established approach for comparing classification methods in the fields of machine learning and data mining.

Another possible source of bias are the choices of classifiers explored by this study. There are many such methods and any single study can only use a subset of them. We have chosen representative methods from each major type of traditional machine learning methods. The selected methods cover six out of seven of the categories identified by the recent review from Malhotra [2]. Further work might explore neural networks and evolutionary algorithms which we excluded due to the complexities of parameterization and execution time. Hence, we would encourage other researchers to repeat our study with other classifier learning methods.

6 Conclusions

In this paper, we have reported a comprehensive experiment to explore the effect of using imbalanced learning algorithms when seeking to predict defect-prone software components. This has explored the complex interactions between type of imbalanced learner, type of classifier and input metrics over 27 software defect data sets from the public domain.

Specifically, we have compared 16 different types of imbalanced learning algorithm — along with the control case of no imbalanced learning — with seven representative classifier learning methods (C4.5, RF, SVM, Ripper, *k*NN, LR and NB) using seven different types of input metric data over 27 data sets. Our factorial experimental design yields 22491 combinations. Each combination was evaluated by 10×10-fold cross validation.

We believe our results are valuable for the software engineering *practitioners* for at least three reasons.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

First, our experimental results show a clear, negative relationship between the imbalance ratio and the performance of traditional learning. Though the level of imbalance in most software defect data is not as high as expected, moderate level of imbalance is enough to impact the performance of traditional learning. This means that if your training data shows moderate or worse imbalance do not necessarily expect good defect prediction performance.

Second, imbalanced learning algorithms can ameliorate this impact, particularly if the imbalance level is moderate or higher. However, the unthinking application of any imbalanced learner in any setting is likely to only yield a very small, if any, positive effect.

Third, negative results can be considerably improved through the right choice of classifier and imbalanced learning methods in the context. In contrast, different choices of input metric have little impact upon the *improvement* that accrue from imbalanced learning algorithms. Our study has highlighted some strong combinations which are given in the summary Table 14 in particular bagging-based imbalanced ensemble methods and EM1v1.

We also believe there are also additional lessons for *researchers*. First, a number of experimental studies have reported encouraging results in terms of using machine learning techniques to predict defect-prone software units. However, this is tempered by the fact that there is also a great deal of variability in results and often a lack of consistency. Our experiment shows that a significant contributing factor to this variability comes from the data sets themselves in the form of the imbalance ratio.

Second, the choice of a predictive performance measure that enables comparisons between different classifiers is a surprisingly subtle problem. This is particularly acute when dealing with imbalanced data sets which are the norm for software defects. Therefore we have avoided some of the widely used classification performance measures (such as F_1) because they are prone to bias. We have chosen the unbiased performance measure Matthews Correlation Coefficient. Although not the main theme of this study we would encourage fellow researchers to consider unbiased alternatives to the F family of measures [10] or Area Under the Curve [51].

Third, comprehensive experiments tend to be both large and complex which necessitate particular forms of statistical analysis. We advocate use of False Discovery Rate procedures [62] to militate against problems of large numbers of significance tests. We also advocate use of effect size measures [90], with associated confidence limits rather than relying on significance values alone since these may be inflated when the experimental design creates large numbers of observations. In other words highly significant but vanishingly small real world effects may not be that important to the software engineering community.

Finally we make our data and program available to other researchers and would encourage them to confirm (or challenge) the strength of our findings so that we are able to increase the confidence with which we make recommendations to software engineering practitioners. The data and scripts are available from: https://zenodo.org/badge/latestdoi/94171384.

ACKNOWLEDGMENT

We thank the reviewers and associate editor for their many constructive suggestions. This work is supported by the National Natural Science Foundation of China under grants 61373046 and 61210004 and by Brunel University London.

REFERENCES

[1] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, 2012.

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

[3] C. Catal and B. Diri, "Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem," *Information Sciences*, vol. 179, no. 8, pp. 1040–1058, 2009.

[4] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *30th ACM/IEEE International Conference on Software Engineering*. IEEE, 2008, pp. 181–190.

[5] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 419–429.

[6] Y. Shin, A. Meneely, L. Williams, and J. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[7] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[8] Q. Cai, H. He, and H. Man, "Imbalanced evolving self-organizing learning," *Neurocomputing*, vol. 133, pp. 258–270, 2014.

[9] H. He and E. Garcia, "Learning from imbalanced data," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 21, no. 9, pp. 1263–1284, 2009.

[10] D. Powers, "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation," *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63, 2011.

[11] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 603–616, 2014.

[12] G. Weiss, "Foundations of imbalanced learning," *Imbalanced Learning: Foundations, Algorithms, and Applications*, pp. 13–41, 2013.

[13] R. Holte, L. Acker, B. Porter *et al.*, "Concept learning and the problem of small disjuncts." in *IJCAI*, vol. 89. Citeseer, 1989, pp. 813–818.

[14] S. Wang and X. Yao, "Diversity analysis on imbalanced data sets by using ensemble models," in *Computational Intelligence and Data Mining, 2009. CIDM'09. IEEE Symposium on*. IEEE, 2009, pp. 324–331.

[15] L. Kuncheva and J. Rodríguez, "A weighted voting framework for classifiers ensembles," *Knowledge and Information Systems*, vol. 38, no. 2, pp. 259–275, 2014.

[16] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[17] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[18] N. Chawla, K. Bowyer, L. Hall, and P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[19] V. López, A. Fernández, S. García, V. Palade, and F. Herrera, "An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics," *Information Sciences*, vol. 250, pp. 113–141, 2013.

[20] M. Galar, A. Fernandez, E. Barrenechea, H. Bustince, and F. Herrera, "A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 4, pp. 463–484, 2012.

[21] P. Branco, L. Torgo, and R. P. Ribeiro, "A survey of predictive modeling on imbalanced domains," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2:31, 2016.

[22] G. Batista, R. Prati, and M. Monard, "A study of the behavior of several methods for balancing machine learning training data," *ACM SigKDD Explorations Newsletter*, vol. 6, no. 1, pp. 20–29, 2004.

[23] E. Ramentol, Y. Caballero, R. Bello, and F. Herrera, "Smote-rsb*: a hybrid preprocessing approach based on oversampling and undersampling for high imbalanced data-sets using smote and rough sets theory," *Knowledge and information systems*, vol. 33, no. 2, pp. 245–265, 2012.

[24] K. Ting, "An instance-weighting method to induce cost-sensitive trees," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 3, pp. 659–665, 2002.

[25] Z. Qin, A. T. Wang, C. Zhang, and S. Zhang, "Cost-sensitive classification with k-nearest neighbors," in *International Conference on Knowledge Science, Engineering and Management*. Springer, 2013, pp. 112–131.

[26] Y. Sahin, S. Bulkan, and E. Duman, "A cost-sensitive decision tree approach for fraud detection," *Expert Systems with Applications*, vol. 40, no. 15, pp. 5916–5923, 2013.

[27] M. Kukar and I. Kononenko, "Cost-sensitive learning with neural networks," in *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*. John Wiley & Sons, 1998, pp. 445–449.

[28] J. Xu, Y. Cao, H. Li, and Y. Huang, "Cost-sensitive learning of svm for ranking," in *European Conference on Machine Learning*. Springer, 2006, pp. 833–840.

[29] Y. Sun, A. K. Wong, and M. S. Kamel, "Classification of imbalanced data: a review," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 23, no. 04, pp. 687–719, 2009.

[30] J. Kittler, M. Hatef, R. Duin, and J. Matas, "On combining classifiers," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 3, pp. 226–239, 1998.

[31] N. Oza and K. Tumer, "Classifier ensembles: Select real-world applications," *Information Fusion*, vol. 9, no. 1, pp. 4–20, 2008.

[32] X. Wu, V. Kumar, R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip *et al.*, "Top 10 algorithms in data mining," *Knowledge and information systems*, vol. 14, no. 1, pp. 1–37, 2008.

[33] R. Barandela, R. M. Valdovinos, and J. S. Sánchez, "New applications of ensembles of classifiers," *Pattern Analysis & Applications*, vol. 6, no. 3, pp. 245–256, 2003.

[34] C. Seiffert, T. Khoshgoftaar, J. Van Hulse, and A. Napolitano, "Rusboost: A hybrid approach to alleviating class imbalance," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 40, no. 1, pp. 185–197, 2010.

[35] N. Chawla, A. Lazarevic, L. Hall, and K. Bowyer, "Smoteboost: Improving prediction of the minority class in boosting," in *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 2003, pp. 107–119.

[36] Z. Sun, Q. Song, and X. Zhu, "Using coding-based ensemble learning to improve software defect prediction," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 42, no. 6, pp. 1806–1817, 2012.

[37] C. Seiffert, T. Khoshgoftaar, and J. Van Hulse, "Improving software-quality predictions with data sampling and boosting," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 39, no. 6, pp. 1283–1294, 2009.

[38] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with precision: A response to" comments on'data mining static code attributes to learn defect predictors'"," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, 2007.

[39] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of ceiling effects in defect predictors," in *Proceedings of the 4th international workshop on Predictor models in software engineering*. ACM, 2008, pp. 47–54.

[40] C. Seiffert, T. Khoshgoftaar, J. Van Hulse, and A. Folleco, "An empirical study of the classification performance of learners on imbalanced and noisy software quality data," *Information Sciences*, vol. 259, pp. 571–595, 2014.

[41] L. Pelayo and S. Dick, "Applying novel resampling strategies to software defect prediction," in *Fuzzy Information Processing Society, 2007. NAFIPS'07. Annual Meeting of the North American*. IEEE, 2007, pp. 69–72.

[42] N. Seliya, T. Khoshgoftaar, and J. Van Hulse, "Predicting faults in high assurance software," in *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*. IEEE, 2010, pp. 26–34.

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

[44] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto, "The effects of over and under sampling on fault-prone module detection," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 2007, pp. 196–204.

[45] P. Baldi, S. Brunak, Y. Chauvin, C. Andersen, and H. Nielsen, "Assessing the accuracy of prediction algorithms for classification: an overview," *Bioinformatics*, vol. 16, no. 5, pp. 412–424, 2000.

[46] M. Warrens, "On association coefficients for 2×2 tables and properties that do not depend on the marginal distributions," *Psychometrika*, vol. 73, no. 4, pp. 777–789, 2008.

[47] J. Xuan, H. Jiang, Y. Hu, Z. Ren, W. Zou, Z. Luo, and X. Wu, "Towards effective bug triage with software data reduction techniques," *IEEE transactions on knowledge and data engineering*, vol. 27, no. 1, pp. 264–280, 2015.

[48] H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang, and X. Luo, "Rosf: Leveraging information retrieval and supervised learning for recommending code snippets," *IEEE Transactions on Services Computing*, 2016.

[49] F. Provost, T. Fawcett, and R. Kohavi, "The case against accuracy estimation for comparing induction algorithms." in *ICML*, vol. 98, 1998, pp. 445–453.

[50] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006.

[51] D. Hand, "Measuring classifier performance: a coherent alternative to the area under the ROC curve," *Machine Learning*, vol. 77, no. 1, pp. 103–123, 2009.

[52] P. Flach and M. Kull, "Precision-recall-gain curves: PR analysis done right," in *Advances in Neural Information Processing Systems*, 2015, pp. 838–846.

[53] P. Domingos, "Metacost: A general method for making classifiers cost-sensitive," in *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1999, pp. 155–164.

[54] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*. IEEE, 2007, pp. 9–9.

[55] A. Tosun, B. Turhan, and A. Bener, "Validation of network measures as indicators of defective modules in software systems," in *Proceedings of the 5th international conference on predictor models in software engineering*. ACM, 2009, p. 5.

[56] D. Aha, D. Kibler, and M. Albert, "Instance-based learning algorithms," *Machine Learning*, vol. 6, no. 1, pp. 37–66, 1991.

[57] W. Cohen, "Fast effective rule induction," in *Proceedings of the twelfth international conference on machine Learning*, 1995, pp. 115–123.

[58] R. Premraj and K. Herzig, "Network versus code metrics to predict defects: A replication study," in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. IEEE, 2011, pp. 215–224.

[59] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[60] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th International Conference on Software Engineering*. ACM, 2008, pp. 531–540.

[61] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[62] Y. Benjamini and D. Yekutieli, "The control of the false discovery rate in multiple testing under dependency," *Annals of Statistics*, pp. 1165–1188, 2001.

[63] R. Coe, "It's the effect size, stupid: What effect size is and why it is important," 2002.

[64] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, no. 3, pp. 494–509, 1993.

[65] J. D. Long, D. Feng, and N. Cliff, "Ordinal analysis of behavioral data," *Handbook of Psychology*, 2003.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

[66] D. Feng, "Robustness and power of ordinal d for paired data," *Real data analysis*, pp. 163–183, 2007.

[67] J. Romano, J. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohensd indices the most appropriate choices," in *annual meeting of the Southern Association for Institutional Research*, 2006.

[68] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[69] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan, "The promise repository of empirical software engineering data," *West Virginia University, Department of Computer Science*, 2012.

[70] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4–5, pp. 531–577, 2012.

[71] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the nasa software defect datasets," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1208–1215, 2013.

[72] Softlab, "ar 1-6," Feb. 2009. [Online]. Available: https://doi.org/10.5281/zenodo.322460

[73] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 15–25.

[74] K. Herzig, S. Just, A. Rau, and A. Zeller, "Predicting defects using change genealogies," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 2013, pp. 118–127.

[75] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[76] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Software Engineering*, vol. 19, no. 2, pp. 167–199, 2012.

[77] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," *Information and Software Technology*, vol. 59, pp. 170–190, 2015.

[78] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 297–308.

[79] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 432–441.

[80] B. Ghotra, S. McIntosh, and A. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 789–800.

[81] F. Zhang, Q. Zheng, Y. Zou, and A. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 309–320.

[82] A. Okutan and O. T. Yıldız, "Software defect prediction using bayesian networks," *Empirical Software Engineering*, vol. 19, no. 1, pp. 154–181, 2014.

[83] A. Kaur and R. Malhotra, "Application of random forest in predicting fault-prone classes," in *Advanced Computer Theory and Engineering, 2008. ICACTE'08. International Conference on*. IEEE, 2008, pp. 37–43.

[84] S. Shivaji, J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 552–569, 2013.

[85] S. Di Martino, F. Ferrucci, C. Gravino, and F. Sarro, "A genetic algorithm to configure support vector machines for predicting fault-prone components," in *Product-Focused Software Process Improvement*. Springer, 2011, pp. 247–261.

[86] E. Giger, M. D'Ambros, M. Pinzger, and H. Gall, "Method-level bug prediction," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2012, pp. 171–180.

[87] C. Couto, C. Silva, M. Valente, R. Bigonha, and N. Anquetil, "Uncovering causal relationships between software metrics and bugs," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012, pp. 223–232.

[88] N. Chawla, N. Japkowicz, and A. Kotcz, "special issue on learning from imbalanced data sets." *SIGKDD Explorations*, vol. 6, no. 1, pp. 1–6, 2004.

[89] R. Wilcox, *Introduction to robust estimation and hypothesis testing (3rd Edn)*, 3rd ed. Academic Press, 2012.

[90] P. Ellis, *The essential guide to effect sizes: Statistical power, meta-analysis, and the interpretation of research results*. Cambridge University Press, 2010.

[91] R. Batuwita and V. Palade, *Class imbalance learning methods for support vector machines*. Wiley, 2013.

[92] R. Burt, *Structural Holes: The Social Structure of Competition*. Harvard University Press, 1995.

Qinbao Song (1966-2016) received the PhD degree in computer science from Xi'an Jiaotong University, Xi'an, China, in 2001. He is a professor of software technology in the Department of Computer Science and Technology, Xi'an Jiaotong University. He is also an adjunct professor in the State Key Laboratory of Software Engineering, Wuhan University, Wuhan, China. He has authored or coauthored more than 100 refereed papers in the areas of machine learning and software engineering. He is a board member of the Open Software Engineering Journal. His research interests include data mining/machine learning, empirical software engineering, and trustworthy software.

Yuchen Guo received BE degree in information and computational science, from Xi'an Jiaotong University, Xi'an, China. He is currently a Ph.D student in Department of Computer Science and Technology, Xi'an Jiaotong University. He is also a member of BSEL (Brunel Software engineering laboratory) as a visiting student. His research project is refining the experimental design and evaluation for practical prediction systems to defect fault-prone software components.

Martin Shepperd received the PhD degree in computer science from the Open University in 1991 for his work in measurement theory and its application to empirical software engineering. He is a professor of software technology at Brunel University, London, United Kingdom. He has published more than 150 refereed papers and three books in the areas of software engineering and machine learning. He was editor-in-chief of the journal Information & Software Technology (1992-2007) and was an associate editor of the IEEE Transactions on Software Engineering (2000-2004). He is currently an associate editor of the journal Empirical Software Engineering. He was elected fellow of the British Computer Society in 2007.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI10.1109/TSE.2018.2836442,

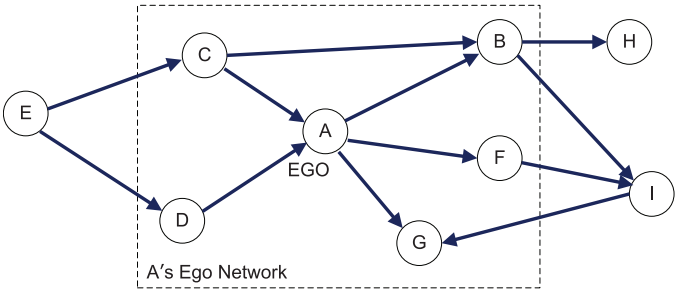

Fig. 9: Ego Network

APPENDIX A
METRIC DEFINITIONS

A.1 CK Metrics

Chidamber-Kemerer (CK) metrics suite [59]:

- *WMC*: Weighted Methods Pr 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

A.2 Network Metrics

A.2.1 Ego network metrics

An ego network is a subgraph that consists of a node (referred to as an "ego") and its neighbours that have a relationship represented by an edge with the "ego" node). This describes how a node is connected to its neighbours, for example, in Fig. 9, node A is the "ego", and the nodes in the box consist A's ego network.

Ego network metrics include:

- *The size of the ego network (Size)* is the number of nodes connected to the ego network.
- *Ties of ego network (Tie)* are directed ties corresponding to the number of edges.
- *The number of ordered pairs (Pairs)* is the maximal number of directed ties, i.e., *Size*×*(Size - 1)*.
- *Density of ego network (Density)* is the percentage of possible ties that are actually present, i.e., *Ties / Pairs*.
- *WeakComp* is the number of weak components (= sets of connected nodes) in neighborhood.
- *nWeakComp* is the number of weak components normalized by size, i.e., *WeakComp / Size*.
- *TwoStepReach* is the percentage of nodes that are two steps away.
- *The reach efficiency (ReachEfficency)* normalizes TwoStepReach by size, i.e., *TwoStepReach / Size*. High reach efficiency indicates that egos' primary contacts are influential in the network.
- *Brokerage* is the number of pairs not directly connected. The higher this number, the more paths go through ego, i.e., ego acts as a brokers in its network.
- *nBrokerage* is the Brokerage normalized by the number of pairs, i.e., *Brokerage / Pairs*.
- *EgoBetween* is the percentage of shortestpaths between neighbors that pass through ego.
- *nEgoBetween* is the Betweenness normalized by the size of the ego network.

A.2.2 Structural metrics

Structural metrics describe the structure of the whole dependency graph by extracting the feature of structural holes, which are suggested by Ronald Burt [92].

- *Effective size of network (EffSize)* is the number of entities that are connected to a module minus the average number of ties between these entities.
- *Efficiency* normalizes the effective size of a network to the total size of the network.
- *Constraint* measures how strongly a module is constrained by its neighbors.
- *Hierarchy* measures how the constraint measure is distributed across neighbors.

A.2.3 Centrality Metrics

Centrality metrics measure position importance of a node in the network.

- *Degree* is the number of edges that connect to a node, which measure dependencies for a module.
- *nDegree* is Degree normalized by number of nodes.
- *Closeness* is sum of the lengths of the shortest paths from a node from all other nodes.
- *Reachability* is the number nodes that can be reached from a node.
- *Eigenvector* assigns relative scores to all nodes in the dependency graphs.
- *nEigenvector* is Eigenvector normalized by number of nodes.
- *Information* is Harmonic mean of the length of paths ending at a node.
- *Betweenness* measures for a node on how many shortest paths between other nodes it occurs.
- *nBetweenness* is Betweenness normalized by the number of nodes.

A.3 Process Metrics

The extracted PROC metrics as suggested by Moser et al. [4] are as follows:

- REVISIONS is the number of revisions of a file.
- AUTHORS is the number of distinct authors that checked a file into the repository.
- LOC_ADDED is the sum over all revisions of the lines of code added to a file.
- MAX_LOC_ADDED is the maximum number of lines of code added for all revisions.
- AVE_LOC_ADDED is the average lines of code added per revision.
- LOC_DELETED is the sum over all revisions of the lines of code deleted from a file.
- MAX_LOC_DELETED is the maximum number of lines of code deleted for all revisions.
- AVE_LOC_DELETED is the average lines of code deleted per revision.
- CODECHURN is te sum of (added lines of code - deleted lines of code) over all revisions.
- MAX_CODECHURN is the maximum CODECHURN for all revisions.
- AVE_CODECHURN is the average CODECHURN per revision.