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 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 standard 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 the performance of imbalanced learning and its complex interactions with (i) data sets (ii) classifiers, (iii) different metrics and (iv) imbalanced learning methods.
Method: We systematically evaluate 27 data sets, 7 classifiers, 7 input metrics and 17 imbalanced learning methods (including doing nothing); 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) Imbalance in software defect data clearly harms the performance of standard learning even if the imbalance ratio is not severe. b) Imbalance learning methods are recommended when the imbalance ratio is greater than 4. c) The particular choice of classifiers and imbalance learning methods is important. d) Though the improvement of imbalanced learning on different input metrics are similar, the performance various a lot.
Conclusion: This paper shows that predicting software defects with imbalanced data can be very challenging. Fortunately the appropriate combination of imbalanced learner and classifier can a good way to ameliorate this problem, but the indiscriminate application of imbalanced learning can be problematic. Other actionable findings include using a wide spectrum of input metrics derived from static code analysis, network analysis and from the development process.

Index Terms—Defect prediction, bug prediction, imbalanced learning, ensemble 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 number and size 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, this 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 through 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 modules 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 it's performance if it were to be used in the 'wild'.

A problem that is frequently encountered is that real world software defect data consists of only a few defective modules (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 skewed, known as imbalanced data in the field of machine learning. When learning from imbalanced data, standard machine learning algorithms struggle [8] and consequently perform poorly in finding rare classes. The

- *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*
- *M. Shepperd is with the Dept. of Computer Science, Brunel University London, UK. E-mail: martin.shepperd@brunel.ac.uk*

Manuscript received June 13, 2017; revised ??.

underlying reasons are that most algorithms:

- suppose 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]. 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 [11]. However, this bias does not work well when generalizing small disjunctive concepts for the minority class [12]. The learning algorithms tend to be overwhelmed by the majority class and ignore the minority class [13], a little like finding proverbial needles in a haystack.

As a result, imbalanced learning has become an active research topic [9], [8], [14] and a number of imbalanced learning methods have been proposed such as bagging [15], boosting [16] and SMOTE [17]. 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, performance measures commonly used are biased. Second, 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 type (e.g., static code or process metrics, network metrics[1]). Third, the relationship between the imbalance ratio and the predictive performance is unexplored for software defect data. Consequently, there is a need to systematically explore the following questions regarding the use and value of imbalanced learning algorithms.

1) How does standard learning perform under imbalanced data?
2) How does imbalanced learning perform compared with standard learning?
3) What is the effect of the following factors: (i) data sets (including imbalance ratio and types of input metric) (ii) type of classifier algorithm (iii) imbalanced learning methods?

This paper makes the following contributions:

1) Given the complexity and contradictory results emerging from other studies we exhaustively evaluate the impact of different classifiers, data sets (imbalance ratio) and input metrics. This is 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) Our experiment is conducted using 27 data sets all in the public domain. This enables us to thoroughly explore impact of imbalance ratio of defect data upon prediction capability and how it can be remediated.
3) We generate a number of practical or actionable findings. These include that we show that imbalanced data is a challenge for software defect prediction. Our findings suggest that imbalanced learners should be deployed if the imbalance ratio exceeds four. We show that the blind application of imbalanced learners may not be successful but that particular combinations of imbalance learner and classifier can yield very practical improvements in prediction.
4) Finally, we demonstrate that typical classification performance measures (e.g., *F-measure* and *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.

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 have 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 four categories:

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

We briefly review these. For more detailed accounts see [9], [18].

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

1. Input metric types are limited which barely include popular software metrics such as process metrics and network metrics for the research of imbalanced learning on software defect prediction

similar way to standard classification [19], [17]. 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 it 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 [18]. *3) SMOTE* [17] is a special over-sampling method that seeks to avoid overfitting by synthetically creating new minority class instances via interpolation between near neighbours . *4) Hybrid* methods combine more than one sub-sampling technique [20].

Cost-sensitive learning can be naturally applied to address imbalanced learning problems [21]. 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 [22], cost-sensitive decision trees [23], cost-sensitive neural networks [24], and cost-sensitive support vector machines [25]. 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 [27]. The most widely used methods are Bagging [15] and Boosting [16] whose applications in several classification problems have led to significant improvements [28]. 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 original data, which focus on difficult instances. AdaBoost [16] is the most commonly used boosting method, and was identified as one of the top ten most influential data mining algorithms [29].

Imbalanced ensemble learning combined ensemble learning with the aforementioned sub-sampling techniques to address the problems of imbalanced data classification. Here the idea is straightforward: 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 that used by Bagging were carried out before training each classifier this leads to UnderBagging [13], OverBagging [30], UnderOverBagging [13], and SMOTE-Bagging [13]. In the same way, by integrating under-sampling and SMOTE with Boosting we obtain RUS-Boost [31] and SMOTEBoost [32]. In instead of sampling, EM1v1 [33] handles the imbalanced data by splitting and coding techniques.

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 [35], 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.

In addition, 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 [10]. 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.

2. Misclassification costs could be given by domain experts, or can be learned via other approaches [26], 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.

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	[34][33][35][36][37][38][39]		[4][40][39]		SVM+Bst					IBk+OBag				
RF	[33][7][37]				SVM+US	[37]				IBk+UOBag				
SVM	[37]				SVM+OS	[37]				IBk+SBag				
Ripper	[33][37]				SVM+UOS					IBk+UBst				
Ibk	[37]				SVM+SMOTE	[37]				IBk+OBst				
LR	[41][37]		[4]		SVM+COS					IBk+UOBst				
NB	[33][7][36][37][39]		[4][39]		SVM+EM1v1					IBk+SBst				
C4.5+Bag	[33]				SVM+UBag					LR+Bag				
C4.5+Bst	[7][34][33]		[40]		SVM+OBag					LR+Bst				
C4.5+US	[34][33][7][36][37]				SVM+UOBag					LR+US	[41][37]			
C4.5+OS	[34][33][36][37]				SVM+SBag					LR+OS	[41][37]			
C4.5+UOS					SVM+UBst					LR+UOS				
C4.5+SMOTE	[33][37][38]				SVM+OBst					LR+SMOTE	[41][37]			
C4.5+COS	[33][7]		[4][40]		SVM+UOBst					LR+COS				
C4.5+EM1v1	[33]				SVM+SBst					LR+EM1v1				
C4.5+UBag	[39]		[39]		Ripper+Bag	[33]				LR+UBag				
C4.5+OBag					Ripper+Bst	[33]				LR+OBag				
C4.5+UOBag					Ripper+US	[33][37]				LR+UOBag				
C4.5+SBag					Ripper+OS	[33][37]				LR+SBag				
C4.5+UBst					Ripper+UOS					LR+UBst				
C4.5+OBst					Ripper+SMOTE	[33][37]				LR+OBst				
C4.5+UOBst					Ripper+COS	[33]				LR+UOBst				
C4.5+SBst	[7]				Ripper+EM1v1	[33]				LR+SBst				
RF+Bag	[33]				Ripper+UBag					NB+Bag	[33]			
RF+Bst	[33]				Ripper+OBag					NB+Bst	[33]			
RF+US	[33][37]				Ripper+UOBag					NB+US	[33][36][37]			
RF+OS	[33][37]				Ripper+SBag					NB+OS	[33][36][37]			
RF+UOS					Ripper+UBst					NB+UOS				
RF+SMOTE	[33][37]				Ripper+OBst					NB+SMOTE	[33][37]			
RF+COS	[33]				Ripper+UOBst					NB+COS	[33]			
RF+EM1v1	[33]				Ripper+SBst					NB+EM1v1	[33]			
RF+UBag					IBk+Bag					NB+UBag	[39]		[39]	
RF+OBag					IBk+Bst					NB+OBag				
RF+UOBag					IBk+US	[37]				NB+UOBag				
RF+SBag					IBk+OS	[37]				NB+SBag				
RF+UBst					IBk+UOS					NB+UBst				
RF+OBst					IBk+SMOTE	[37]				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

First, Menzies et al. [36] conducted an experiment based on twelve 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. [33]. However, Menzies et al. also 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. [37] 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. [36] and Sun et al. [33]. 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. [34] is where they compared sub-sampling methods with Boosting for improving the performance of decision tree model built for identifying the defective modules. Their results show that Boosting outperform even the best sub-sampling methods. In contrast, Khoshgoftaar et al. [40] built software quality models by using Boosting and cost-sensitive Boosting where C4.5 and decision stumps were used as the base classifiers, respectively. They found that Boosting and cost-sensitive Boosting do not enhance the performance of individual pruned C4.5 decision tree.

Therefore, our study focuses on an exhaustive comparison of $16 * 7 = 112$ different popular imbalanced learning methods with seven representative and widely used standard 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.

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 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 unbiased data sets which is of course precisely the scenario we are interested in studying. Consequently, we use the Matthews correlation coefficient *MCC* [42] as our measure of predictive performance.

The starting point for most classification measures is the confusion matrix. This represents the four possible outcomes when using a dichotomous classifier to make a prediction (see Table 2)[4].

	Actually +ve	Actually -ve
Predict Positive	TP	FP
Predict Negative	FN	TN

TABLE 2: Confusion Matrix

F_1 is the most commonly used derivative of the *F-measure* family and is defined by Eqn. 1.

$$F_1 = \frac{2 \cdot \mathrm{TP}}{2 \cdot \mathrm{TP} + \mathrm{FP} + \mathrm{FN}}. \quad (1)$$

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 recommending task[5], this is not so for defect prediction because test managers would be happy to know if components are truly non-defective.

Let us compare F_1 with the Matthews correlation coefficient (*MCC*, also known as ϕ - see [45]). MCC is the geometric mean of the regression coefficients of the problem and its dual [46] and is defined as:

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

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 defect classifier predicts as following:

	Actually +ve	Actually -ve
Predict Positive	5	45
Predict Negative	5	45

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.

	Actually +ve	Actually -ve
Predict Positive	5	45
Predict Negative	5	0

Now suppose the True Negatives are removed so n=55. 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.

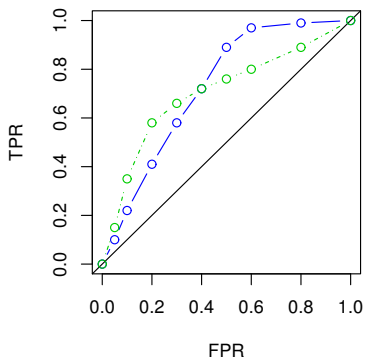
This example illustrates not only the drawback of F_1, but also the weakness of all derivative measures from *Recall* and *Precision* as they ignore TNs. Measures such as *Accuracy* and the *F-measure* are also known to be biased as they are sensitive to data distributions and the prevalence of the positive class [47]. Thus, we seek a measure that satisfies the following requirements:

1) A single metric to cover the whole confusion matrix because one can always be optimized at the expense of the other we seek a single metric to compare classifiers;
2) Easy to interpret so in our case aligned from plus unity for a perfect classifier, through zero for no association, i.e., random performance to minus unity for a perfectly perverse classifier;
3) Properly takes into account the underlying frequencies of true and negative cases;
4) Evaluates a specific classifier, as opposed to a family of classifiers such as is the case for the Area Under the Curve (AUC) measure [48]

The fourth 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.

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

5. Recommending is in the information retrieval domain, such as bug triage [43] or recommending code snippets [44].



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

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

TABLE 3: Points on the A and B ROC curves

Consider the example in Fig. 1 that shows ROC curves for two classifiers (Classifier Family A and Classifier Family B) derived from the values of some points on these curves (Table 3). 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 [49], 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.

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 4) to seven standard classifiers chosen to be representative of commonly used approaches [1] (see Table 5). We then use seven classes of input metric (see Table 6). 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 for a given data set.

Then for each combination we use M×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 data sets are randomized. In our case $M = 10$ and $N = 10$ so overall, 100 models are built and 100 results obtained for each data set.

To summarize, the experimental process is shown by the following pseudo-code. Notice that attribute selection is applied to the training data of each base learner, see Lines 14 and 22.

Method Type	Abbr	Method Name	Ref
Sub-sampling methods	US	Under-Sampling	[17]
	OS	Over-Sampling	[17]
	UOS	Underover-Sampling	[17]
	SMOTE	SMOTE	[17]
Cost-sensitive methods	COS	Cost-sensitive learning	[50]
Ensemble methods	Bag	Bagging	[15]
	Bst	Boosting	[16]
Imbalanced ensemble methods	EM1v1	EM1v1	[33]
	UBag	UnderBagging	[13]
	OBag	OverBagging	[30]
	UOBag	UnderoverBagging	[13]
	SBag	SMOTEBagging	[13]
	UBst	UnderBoosting	[31]
	OBst	OverBoosting	
	UOBst	UnderoverBoosting	
	SBst	SMOTEBoosting	[32]

TABLE 4: Summary of imbalanced learning methods

Abbr	Classification Algorithm	Ref
LR	Logistic Regression	[6], [51]
NB	Naïve Bayes	[52]
C4.5	Decision tree	[4]
IBk	Instance based *k*NN	[53]
Ripper	Rule based Ripper	[54]
SVM	Support vector machine (SMO)	[55]
RF	Random Forest	[7]

TABLE 5: Summary of classifiers

3.3 Statistical Methods

Given the performance estimates of each classifier on every dataset, how to determinate which classifier is

Input Metrics	Metrics Type	Ref
CK	Source Code metrics	[56]
NET	Network metrics	[57]
PROC	Process metrics	[4]
CK+NET	Combined metrics	-
CK+PROC	Combined metrics	-
NET+PROC	Combined metrics	-
CK+NET+PROC	Combined metrics	-

TABLE 6: Input metric classes used in our experiment

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 standard 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 standard
                   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;
```

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 [58]. 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 Cd_1 obtains better, equal, and worse performance than Classifier Cd_2.

Next, effect size is computed since it emphasises the size of the difference rather than confounding this with sample size [59]. The effect statistics of difference (average improvement) and dominance (Cliff's δ) are both reported. Cliff's δ is a non-parametric robust indicator which measure the magnitude of dominance as the difference between two groups [60]. It estimates the likelihood of how often Predictor Cd_1 is better than Predictor Cd_2. We use the paired version since our data are correlated [61], [62]. By convention, the magnitude of the difference is considered trivial ($|\delta| < 0.147$), small ($0.147 <= |\delta| < 0.33$), moderate ($0.33 <= |\delta| < 0.474$), or large ($|\delta| >= 0.474$) as suggested by Romano et al. [63].

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 [64], network metrics may have a stronger association with defects [57] and process metrics reflect the changes to software systems over time. We also consider combinations of these metrics yielding a total of seven possibilities (Table 6). The details are as follows:

(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 [56] which are 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 [57]. In this study, the networks are call graphs of software systems, where the nodes are the components of a software and the edges are the call dependencies among these components. The DependencyFinder[6] tool was used to extract the call relations. Once networks are built, the UCINET[7] tool was employed to calculate three kinds of network (NET) metrics of dependency networks, i.e., 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

The PROMISE repository [65] includes many software defect prediction data sets that are publicly available and widely used by many researchers from which we selected 22 data sets. To this we added a further 5 from D'Ambros et al.'s defect prediction benchmark data sets [66]. Thus we use a total of 27 data sets 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 7). From these data sets we extract the necessary metrics and also calculate the imbalance ratio

6. http://depfind.sourceforge.net/
7. http://www.analytictech.com/ucinet/

(IR[8]). These show considerable diversity ranging from 1.51 to 43.73.

Note that although the NASA MDP data sets have been widely used in developing defect prediction models we do not use them because "although the repository holds many metrics and is publicly available, it does have limitations. It is not possible to explore the source code and the contextual data are not comprehensive (e.g., no data on maturity are available). It is also not always possible to identify if any changes have been made to the extraction and computation mechanisms over time. In addition, the data may suffer from important anomalies." [1].

4 Experimental Results and Analysis

In this section, we first look at the basic characteristics of imbalanced learning problem on software defect prediction by investigating:

RQ1) How does the level of predictive performance vary?
RQ2) How does the standard learning perform under imbalance?
RQ3) How does the imbalanced learning perform compare with standard learning?
RQ4) What is the relationship between the imbalance ratio (IR) and the effect of imbalanced learning?

Then we explore the effect of imbalanced learning and its complex interactions with our three experimental factors by answering the following questions:

RQ5) How do classifiers matter?
RQ6) How do input metrics matter?
RQ7) How do imbalanced learning methods matter?

4.1 Variation in predictive performance

As we discussed in Section 3.1 our choice of a measure of predictive performance (i.e., our response variable) is *MCC*. It is unbiased and is easy to interpret as a correlation coefficient (+1 denotes perfect classification, 0 no association between predicted and actual and -1 a perfectly perverse classification).

Table 8 provides some basic summary statistics for our response variable. First, we observe considerable spread from a maximum of 0.679 to a disappointing -0.112. Note that negative values indicate perverse performance and an immediate improvement could be achieved by doing the opposite of what the classifier predicts! Next it can be seen that there is negative skewness and the median exceeds the mean. The kurtosis is 2.67 which suggests rather fat tails, again confirmed by visual inspection of the histogram and also of the qqplot in Fig. 2 where we see the tails deviating from the expected normal distribution shown as a red line. This suggests the need for non-parametric and robust statistical techniques [79]

8. IR is defined as the ratio of the number of the majority class instances to the number of the minority class instances [67]

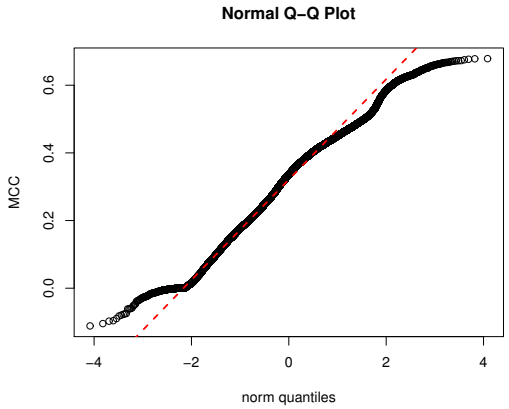


Fig. 2: QQplot of predictive performance (MCC)

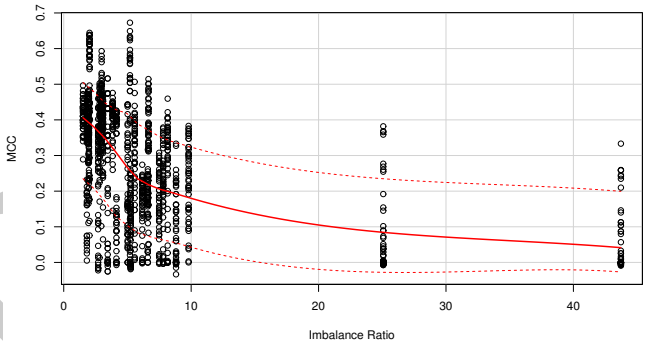


Fig. 3: Performance of predictive performance (MCC) without imbalanced learning

So the answer for *RQ1* is that the distribution of predictive performance (MCC) departs from normality at a level that cannot be ignored. This would indicate that parametric techniques should be used with caution and robust statistics are to be preferred.

4.2 Performance of Standard Learning

To understand what happens if we only employ standard learning, we show the relationship between the imbalance ratio and the predictive performance (MCC) without imbalanced learning (see Fig. 3). The red line is drawn by a non-parameter smoother (loess smoothing) and the dotted lines indicate ± one standard deviation.

From this enhanced scatter plot we observe that imbalance ratio has a negative impact upon classification performance; broadly speaking the more imbalanced the data set the worse the prediction. Although most observations are located in the range [1,10] of IR (excepting the observations from two extremely imbalanced data sets), the smoothed line reduces rapidly. It can be seen that even a small increase in the imbalance ratio can potentially cause a substantial reduction in predictive performance even the imbalance ratio is not high.

Data	Modules	Defective Modules	IR	References	Data	Modules	Defective Modules	IR	References
ant-1.3	125	20	5.25	[68], [69]	log4j-1.0	135	34	2.97	[70], [71]
ant-1.4	178	40	3.45	[68], [69]	poi-2.0	314	37	7.49	[68], [69]
ant-1.5	293	32	8.16	[68], [69]	synapse-1.0	157	16	8.81	[68], [69]
ant-1.6	351	92	2.82	[68], [69]	synapse-1.1	222	60	2.7	[68], [69]
camel-1.0	339	13	25.08	[68], [69]	synapse-1.2	256	86	1.98	[68], [72], [69]
camel-1.2	608	216	1.81	[68], [69]	velocity-1.6	229	78	1.94	[68], [72], [69]
camel-1.4	872	145	5.01	[68], [73], [69]	xerces-1.2	440	71	5.20	[68], [69], [74]
camel-1.6	965	188	4.13	[68], [73], [69] [70], [71]	xerces-1.3	453	69	5.57	[68], [69], [74] [70], [71]
ivy-2.0	352	40	7.8	[68], [72], [69]	Eclipse JDT Core-3.4	997	206	3.84	[66], [75], [74]
jedit-3.2	272	90	2.02	[68], [69], [76] [77]	Eclipse PDE UI-3.4.1	1497	209	6.16	[66], [75],[71]
jedit-4.0	306	75	3.08	[78], [68], [69] [76], [77], [70]	Equinox framework-3.4	324	129	1.51	[66], [75],[71]
jedit-4.1	312	79	2.95	[78], [76], [77]	Lucene-2.4.0	691	64	9.80	[66], [68], [75] [73], [69], [72] [74], [70],[71]
jedit-4.2	367	48	6.65	[78], [76], [77]	Mylyn-3.1	1862	245	6.6	[66],[71]
jedit-4.3	492	11	43.73	[78], [72], [76] [77]	**Mean**	**496.63**	**88.63**	**6.91**	

TABLE 7: Description of the 27 Data Sets

Statistic	Value	Statistic	Value
Min	-0.112	Max	0.679
Mean	0.319	Median	0.335
sd	0.137	Trimmed (0.2) sd	0.140
Skewness	-0.271	Kurtosis	2.672

TABLE 8: Summary statistics for predictive performance (MCC)

The robust percentage bend correlation coefficient [79] is -0.524 with $p < 0.0001$. This indicates a moderate negative correlation between the performance and IR which confirms the smoothing line. Therefore, the answer to *RQ2* is that the performance of standard learning is highly threatened by the imbalance of defect data. It is clear that any means of addressing imbalance in the data is potentially important for software defect prediction.

4.3 Does imbalanced learning help defect prediction?

Our next research question explores whether there is an effect on defect prediction through applying imbalanced learning algorithms. An effect means a non-zero difference between imbalanced learning and standard learning and of course we are most interested in positive effects, i.e., a greater correlation. The data are correlated or paired as a result of the repeated measure design of the experiment thus 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 (difference) as a histogram. The shaded bars in the histogram indicate negative effects, i.e., the imbalanced learning makes the predictive performance worse. Overall, this happens in about 29% of the cases. Careful examination of these negative cases suggests that imbalanced learning can be counter-productive due to the lack of structure to learn from for challenging datasets.

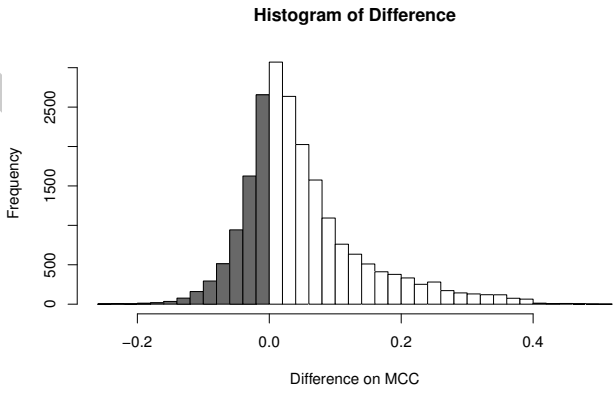


Fig. 4: Histogram of differences in predictive performance (MCC) with and without imbalanced learning

Statistic	Value	Statistic	Value
Min	-0.258	Max	0.504
Mean	0.050	Trimmed mean	0.033
sd	0.092	Trimmed (0.2) sd	0.069
Skewness	1.246	Kurtosis	5.041

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

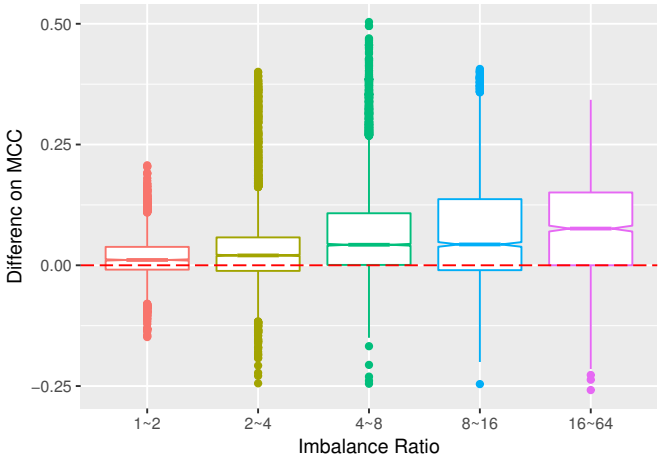


Fig. 5: Boxplot for differences in performance with and without imbalanced learning on 5 imbalance levels

Table 9 provides summary statistics of the differences. We see that the standard deviation (sd) and trimmed sd are both larger than mean or median. 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 is considerably greater than 3 which implies a very heavy tailed distribution. This all indicates that our analysis needs robust statistics. Therefore using the percentile bootstrap method [79] we can estimate the 95% confidence limits around the trimmed mean 0.033 as being (0.032, 0.034). We can be confident the average difference between using an imbalanced learner and 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, there is a great deal of variance must not be ignored.

Interestingly these results are not limited to software defects. Lopez et al. [67] 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 dataset shift. They also pointed out that a reduction of performance could happen 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 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.

4.4 Effect of the Imbalance Ratio

Next, we present in Fig. 5 the distributions of effect (difference) as boxplots grouped into five bins ranging from extremely imbalanced to almost balanced (the rationale for the bins is given in Fig. 6). The notches show the 95% confidence intervals around the median (shown as a thick bar).

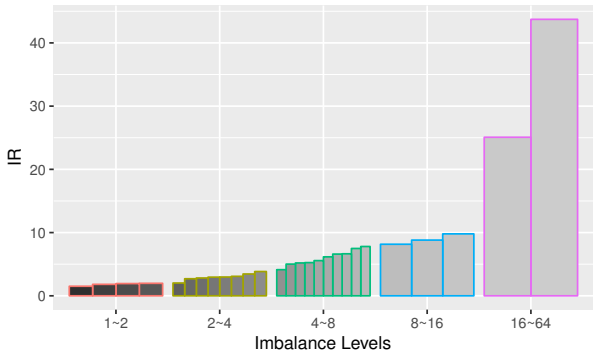


Fig. 6: Data sets' IR in 5 imbalance levels

We observe IR impacts the predictive difference of performance such that the benefits of using an imbalanced learner are greatest when the Imbalance Ratio is most extreme. This is to be expected since the standard learning is predicated on learning from an imbalanced distribution. For each bin, Table 11 presents the effect size as both average improvement (difference in MCC) and dominance statistics which show the stochastic likelihood that imbalanced learning is better than nothing. As shown in the table the higher IR is, the greater improvement the imbalanced learning can gain, which confirms Fig. 5. Also we are confident that the improvement is larger than 0.046 when IR>4 with a non-small Cliff's δ that could interpret a good change to get improved by imbalanced learning, which is the case imbalanced learning should be considered.

IR	Average Improvement	Cliff's δ
1∼2	0.013 (0.011, 0.014)	+S (+S +S)
2∼4	0.022 (0.020, 0.023)	+M (+S +M)
4∼8	0.048 (0.046, 0.050)	+L (+L +L)
8∼16	0.053 (0.048, 0.058)	+M (+M +M)
16∼64	0.076 (0.070, 0.082)	+L (+M +L)

TABLE 10: Effect size (and confidence interval) by imbalance ratio bin. Direction is denoted by + or -. For Cliff's δ, size is denoted as follows: N=negligible, S=small, M=medium, L=large

The overall, robust correlation coefficient[9] is $(0.216, p < 0.0001)$ which indicates non-zero but weak association between IR and improvement as the answer for *RQ4*.

4.5 Effect of Classifier Type

In our experiment we investigate seven different types of classifier (listed in Table 5). Fig. 7 shows the difference i.e., the effect achieved by introducing an imbalanced learning algorithm as boxplots organised by classifier type. Support vector machines (SVM) consistently benefit from imbalanced learning. This is in line with

9. We use the percentage bend correlation coefficient from pbcor in the WRS2 R package.

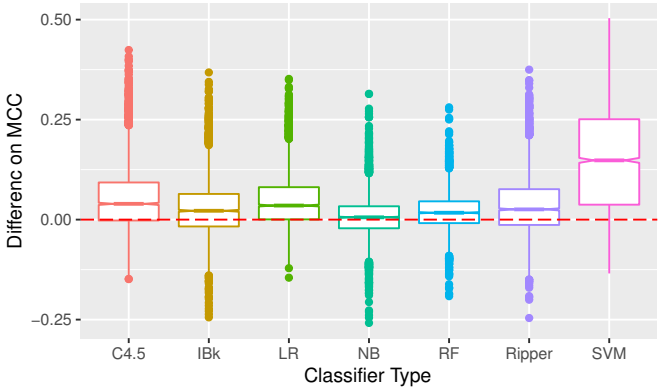


Fig. 7: Boxplot of differences in performance (MCC) with and without imbalanced learning by classifiers

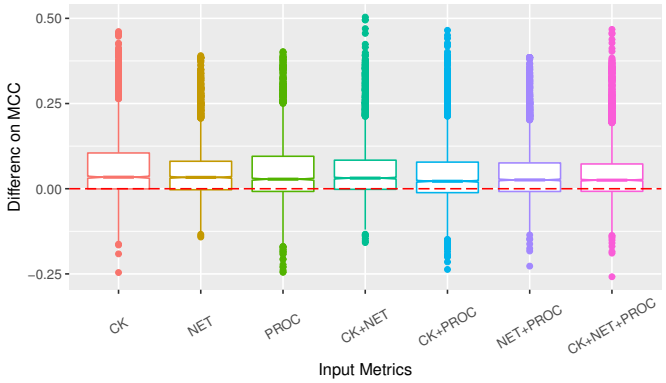


Fig. 8: Boxplot of differences in performance (MCC) with and without imbalanced learning by metrics

Batuwita and Palade [80] 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". Otherwise, there is little evidence of any consistent positive effect, particularly for Naïve Bayes (NB) classifiers which do not appear sensitive to imbalance. 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 to gain improvement.

Classifier	Average Improvement	Cliff's δ
SVM	0.145 (0.140 0.151)	+L (+L +L)
C4.5	0.042 (0.040, 0.045)	+M (+M +L)
LR	0.038 (0.036, 0.040)	+L (+L +L)
Ripper	0.028 (0.026, 0.031)	+S (+S +M)
IBk	0.023 (0.021, 0.025)	+S (+S +M)
RF	0.018 (0.017, 0.020)	+M (+S +M)
NB	0.006 (0.004, 0.007)	+N (+N +S)

TABLE 11: Effect size compared with no imbalanced learning by algorithm

Table 11 presents the effect size of both two statistics for each type of classifier in the decreasing order of the average improvement. 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 imbalanced distribution therefore the effect size on NB is very small. So as for *RQ5*, classifiers' sensitivity to imbalance largely influence the effect size.

4.6 Effect of Input Metrics

The next factor in our experiment is the type of input metric. Seven different classes are summarized in Table 6 and the distributions of the difference on predictive performance through imbalanced learning shown as boxplots grouped by Metrics in Fig. 8. The red dotted line shows zero difference or no effect. Overall we see a high level of similarity between the boxplots and also the effect size of each type of input metric is not significantly different (see Table 12). This indicates little difference in responsiveness to imbalanced learning by type of input metric, which is the answer for *RQ6*.

Input Metrics	Average Improvement	Cliff's δ
CK	0.043 (0.039, 0.046)	+M (+M +L)
NET	0.035 (0.033, 0.038)	+M (+M +M)
PROC	0.036 (0.033, 0.039)	+M (+M +M)
CK+NET	0.035 (0.033, 0.038)	+M (+M +L)
CK+PROC	0.027 (0.024, 0.030)	+M (+M +M)
NET+PROC	0.030 (0.027, 0.032)	+M (+M +M)
CK+NET+PROC	0.029 (0.027 0.032)	+M (+M +M)

TABLE 12: Effect size compared with no imbalanced learning by input metrics

However, in passing we do note that there is considerable difference in *overall* predictive performance depending upon the class of input metric (see Fig. 9. 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).

4.7 Detailed comparisons of imbalanced learner algorithms

Next we review the impact by specific, imbalanced learning algorithm. Fig. 10 shows side by side boxplots for the difference each algorithm makes over no algorithm. The red dashed line shows zero difference. It can be seen that all types of imbalanced methods are capable of producing negative impacts upon 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).

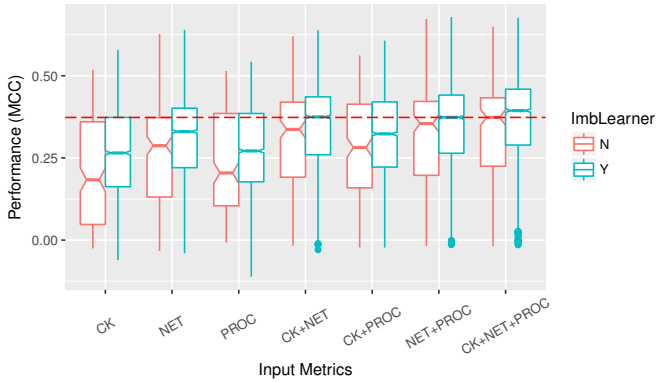


Fig. 9: Boxplot of performance (MCC) with and without imbalanced learning by metrics

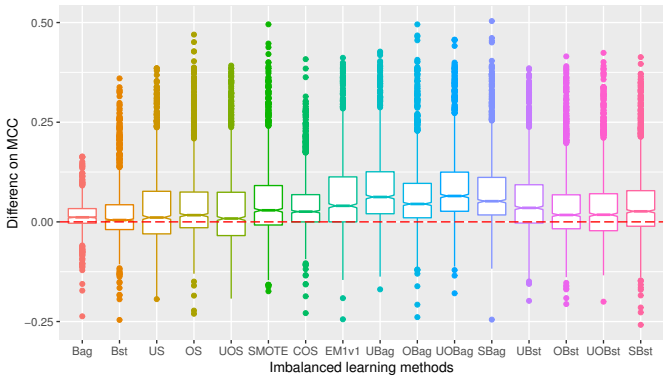


Fig. 10: Boxplot of differences in performance (MCC) with and without imbalanced learning by imbalanced learner

Additionally, Table 13 shows the average improvement. The values in parentheses give the lower and upper bounds of 95% confidence limits. The algorithms are organised in decreasing order of improvement ranging from 0.069 to 0.008. The five types of imbalanced methods that show the largest positive effect in both improvement and Cliff's δ are UOBag, UBag, SBag, OBag and EM1v1.

Combining all the previous analyses Table 14 summarises the results broken down by imbalanced learning algorithm, classifier and input metrics as win/draw/loss counts from the 27 data sets in our experiment. We then use the Benjamini-Yekutieli step-up procedure [58] to determine significance[10]. This is indicated by the graying out of the *non-significan*t cells.

Table 14 summarizes the Win/Draw/Loss (*W/D/L*) records of comparisons between imbalanced learners in the first row and standard learners in the first column over the seven different types of metric data shown in the second column. Each cell contains three counts *W/D/L* for the imbalanced learner against no learning.

10. 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 a more modern approach based on a false discovery rate than other more conservative corrections such as Bonferroni.

Classifier	Improvement	Cliff's δ
UOBag	0.069 (0.065, 0.074)	+L (+L,+L)
UBag	0.067 (0.063, 0.072)	+L (+L +L)
SBag	0.058 (0.054, 0.062)	+L (+L,+L)
OBag	0.049 (0.045, 0.054)	+L (+L,+L)
EM1v1	0.047 (0.043, 0.052)	+L (+L,+L)
UBst	0.039 (0.035, 0.044)	+M (+M,+L)
SMOTE	0.034 (0.030, 0.039)	+M (+M,+M)
COS	0.030 (0.027, 0.033)	+L (+M,+L)
SBst	0.030 (0.026, 0.034)	+M (+S,+M)
OS	0.023 (0.019, 0.027)	+S (+S,+S)
OBst	0.022 (0.018, 0.026)	+S (+S,+S)
UOBst	0.022 (0.018, 0.026)	+S (+S,+S)
US	0.017 (0.012, 0.021)	+N (+N,+S)
Bag	0.013 (0.011, 0.015)	+M (+S,+M)
UOS	0.013 (0.008, 0.018)	+N (+N,+N)
Bst	0.008 (0.005, 0.011)	+N (+N,+S)

TABLE 13: Effect size compared with no imbalanced learning by imbalanced method

iM uses standard learner S as its base learner obtained better, equal, or worse performance outcomes than S using MCC (the performance measure introduced in Section 4). If iM wins S more than loses, a Wilcoxon test is used to verify the assumption iM is better than S. A table cell is shaded if iM does not win S more than lose or iM is not significantly better than S.

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

1) There is greater variability in the performance of the imbalanced learning algorithms compared with standard learner. 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) Approximately half of the table cells are unshaded. This indicates if the imbalanced learning algorithm, classifier and input metrics can be carefully chosen, there are good opportunities to improve predictive performance.
3) The choice of imbalanced learning algorithm strongly depends upon the base classification algorithm. For instance, all the table cells of SVM are unshaded except five cells in the COS column. This indicates that all 16 learner types, excluding COS, can improve SVM for all input metrics. By contrast, none of these algorithms can improve NB as almost all table cells for the NB row are shaded. This supports the idea that NB is insensitive to imbalanced data distribution and therefore is hard to be improve.

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

Note:
(1) Each cell contains the W/D/L counts of each dataset so $W + D + L = 27$.
(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

4) The column of Bagging-based imbalanced ensemble methods (UBag, OBag, UOBag, SBag) and Em1v1 have the largest white area in Table 14 showing that they are clearly better than other imbalanced learners for all rows of C4.5, RF, SVM, Ripper, IBk and LR. We can see that UBag and EM1v1 are the top two of the 16 imbalanced learner types indicated by the largely white columns. Therefore these imbalanced algorithms could be recommended in most circumstances.
5) Bag is not as good as the Bagging-based imbalanced ensemble methods in that it contains more shaded table cells in its column. Similarly Bst is less effective than the Boosting-based imbalanced ensemble methods. The reason behind this is that ensemble learning methods Bagging and Boosting aim to approximate a real target from different directions; they are not inherently designed to deal with imbalanced data. Sub-sampling methods fare a little better. The underlying reason may relate to the random error of sampling. But SMOTE performed better probably because it brings novel information by interpolating between existing ones data points.

As Table 14 illustrates, we can answer RQ7 by noting the 16 imbalanced learning methods vary a lot, however Bagging-based imbalanced ensemble methods and EM1v1 appear to be the top methods.

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. These data sets are characterized by (i) they are drawn from different types of 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 potential replication and comparison with findings from other experiments. Moreover, we have made our raw results and scripts available[11]. However,

11. https://github.com/yuchen1990/ImbSDP

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 choice 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 standard 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 / 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, 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.

First, our experimental results show a clear, negative relationship between the imbalance ratio and the performance of standard learning. This means irrespective of other factors, the more imbalanced your data the more challenging it will be to achieve high quality predictions.

Second, imbalanced learning algorithms can ameliorate this effect, particularly if the imbalance ratio exceeds four. However, the unthinking application of any imbalanced learner in any setting is likely to only yield a very small, if any, positive effect. However, this can be considerably optimized through the right choice of classifier and imbalanced learning methods in the context. Our study has highlighted some strong combinations which are given in the summary table 14 in particular bagging-based imbalanced ensemble methods and EM1v1.

Third, although different choices of input metric have little impact upon the *improvement* that accrue from imbalanced learning algorithms, we have observed they have a very considerable effect upon overall performance. Consequently we recommend, wherever possible, using a wide spectrum of input metrics derived from static code analysis, network analysis and from the development process.

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 [46] or Area Under the Curve [49].

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 [58] to militate against problems of large numbers of significance tests. We also advocate use of effect size measures [81], 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.

Appendix A
Metric Definitions

A.1 CK Metrics

Chidamber-Kemerer (CK) metrics suite [56]:

- *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

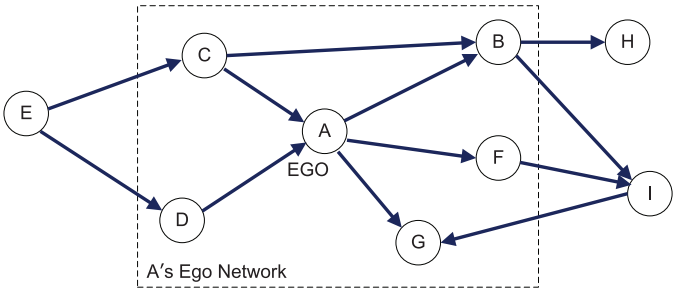


Fig. 11: Ego Network

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 represnted by an edge with the "ego" node). This describes how a node is connected to its neighbours, for example, in Fig. 11, 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 broker 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 [82].

- *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.

ACKNOWLEDGMENT

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

[10] 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.

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

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

[13] 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.

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

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

[16] 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.

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

[18] 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.

[19] G. Batista, R. C. Prati, and M. C. 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.

[20] 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.

[21] K. M. 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] X. Wu, V. Kumar, J. 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.

[30] 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.

[31] C. Seiffert, T. M. 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.

[32] N. V. Chawla, A. Lazarevic, L. O. Hall, and K. W. 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.

[33] 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.

[34] C. Seiffert, T. M. 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.

[35] 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, p. 637, 2007.

[36] 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.

[37] C. Seiffert, T. M. 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.

[38] 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.

[39] N. Seliya, T. M. 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.

[40] T. M. 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] F. J. Provost, T. Fawcett, and R. Kohavi, "The case against accuracy estimation for comparing induction algorithms." in *ICML*, vol. 98, 1998, pp. 445–453.
[48] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006.
[49] D. J. Hand, "Measuring classifier performance: a coherent alternative to the area under the ROC curve," *Machine Learning*, vol. 77, no. 1, pp. 103–123, 2009.
[50] 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.
[51] 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.
[52] 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.
[53] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Machine Learning*, vol. 6, no. 1, pp. 37–66, 1991.
[54] W. W. Cohen, "Fast effective rule induction," in *Proceedings of the twelfth international conference on machine Learning*, 1995, pp. 115–123.
[55] 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.
[56] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.
[57] 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.
[58] Y. Benjamini and D. Yekutieli, "The control of the false discovery rate in multiple testing under dependency," *Annals of Statistics*, pp. 1165–1188, 2001.
[59] R. Coe, "It's the effect size, stupid: What effect size is and why it is important," 2002.
[60] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, no. 3, pp. 494–509, 1993.
[61] J. D. Long, D. Feng, and N. Cliff, "Ordinal analysis of behavioral data," *Handbook of Psychology*, 2003.
[62] D. Feng, "Robustness and power of ordinal d for paired data," *Real data analysis*, pp. 163–183, 2007.
[63] J. Romano, J. D. 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.
[64] 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.
[65] 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.
[66] 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.
[67] 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.
[68] 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.
[69] 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.
[70] B. Ghotra, S. McIntosh, and A. E. 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.
[71] F. Zhang, Q. Zheng, Y. Zou, and A. E. 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.
[72] A. Okutan and O. T. Yıldız, "Software defect prediction using bayesian networks," *Empirical Software Engineering*, vol. 19, no. 1, pp. 154–181, 2014.
[73] 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.
[74] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2012, pp. 171–180.
[75] C. Couto, C. Silva, M. T. 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.
[76] 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.
[77] S. Shivaji, E. 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.
[78] 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.
[79] R. Wilcox, *Introduction to robust estimation and hypothesis testing (3rd Edn)*, 3rd ed. Academic Press, 2012.
[80] R. Batuwita and V. Palade, *Class imbalance learning methods for support vector machines*. Wiley, 2013.
[81] P. D. Ellis, *The essential guide to effect sizes: Statistical power, meta-analysis, and the interpretation of research results*. Cambridge University Press, 2010.
[82] R. S. Burt, *Structural Holes: The Social Structure of Competition*. Harvard University Press, 1995.

Qinbao Song (1966-2016) received the PhD degree in computer science from Xian Jiaotong University, Xian, China, in 2001. He is a professor of software technology in the Department of Computer Science and Technology, Xian Jiaotong University. He is also with 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 current research interests include data mining/machine learning, empirical software engineering, and trustworthy software.

Yuchen Guo received BE degree in information and computational science, from Xian Jiaotong University, Xian, China. He is currently a Ph.D student in Department of Computer Science and Technology, Xian Jiaotong University. He is also currently a member of BSEL (Brunel Software engineering laboratory) as a visiting student. His research area is software defect prediction.

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 program chair for Metrics 01 and 04 and ESEM 11.