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 varies 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 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
underlying reasons are that most algorithms:

- suppose balanced class distributions or equal mis-
classification costs [9], thus fail to properly represent
the distributive characteristics of the imbalanced
data.
- are frequently designed, tested, and optimized ac-
cording 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 imbal-
canced 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 un-
certainty concerning the use of imbalanced learning for
software defect prediction. First, performance measures
commonly used are biased. Second, the interaction be-
 tween 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 metrics1). 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 com-
pared 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 eval-
uate the impact of different classifiers, data sets
(imbalance ratio) and input metrics. This is the
largest single experimental investigation of imbal-
canced 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 re-
mediated.
3) We generate a number of practical or actionable
findings. These include that we show that im-
balanced 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 AUIC)
are unsound and demonstrate a practical alterna-
tive in the form of the Matthews correlation coef-
ficient (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 ma-
chine 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 con-
struction 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 oversampling 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.

**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, undersampling, 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 undersampling and SMOTE with Boosting we obtain RUSBoost [31] and SMOTEBest [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> + <imbalance learner> for instance NB+SMOTE denotes Naïve Bayes coupled with SMOTE. Next there are four³ 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.
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 presents the four possible outcomes when using a dichotomous classifier to make a prediction (see Table 2).

<table>
<thead>
<tr>
<th></th>
<th>Actually +ve</th>
<th>Actually -ve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predict Positive</td>
<td>TP</td>
<td>FP</td>
</tr>
<tr>
<td>Predict Negative</td>
<td>FN</td>
<td>TN</td>
</tr>
</tbody>
</table>

TABLE 2: Confusion Matrix

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

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

However, it excludes True Negatives (TN) in its calculation which is potentially problematic. The reason is that it originated from the information retrieval domain where typically the number of true negatives, e.g., irrelevant web pages that are correctly not returned is neither knowable nor interesting. However unlike recommending task\(^4\), 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 \( \phi \) - see [45]). MCC is the geometric mean of the regression coefficients of the problem and its dual [46] and is defined as:

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

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

Suppose our defect classifier predicts as following:

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].

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.

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.

F-measure

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.
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.

### 3.3 Statistical Methods

Given the performance estimates of each classifier on every dataset, how to determine which classifier is
better? First we need to examine whether or not the performance difference between two predictors could be caused by chance. We use a Wilcoxon signed-rank test (a non-parametric statistical hypothesis test used when comparing paired data) to compare pairs of classifiers. Like the sign test, it is based on difference scores, but in addition to analyzing the signs of the differences, it also takes into account the magnitude of the observed differences. The procedure is non-parametric so no assumptions are made about the probability distributions, which is important since a normal distribution is not always guaranteed. We correct for multiple tests by using the Benjamini-Yekutieli step-up procedure to control the false discovery rate [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 $C_{d_1}$ obtains better, equal, and worse performance than Classifier $C_{d_2}$.

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 $d$) are both reported. Cliff’s $d$ 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 $C_{d_1}$ is better than Predictor $C_{d_2}$. We use the paired version since our data are correlated [61], [62]. By convention, the magnitude of the difference is considered trivial ($|d| < 0.147$), small ($0.147 \leq |d| < 0.33$), moderate ($0.33 \leq |d| < 0.474$), or large ($|d| \geq 0.474$) 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 tool was used to extract the call relations. Once networks are built, the UCINET 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.
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 $\text{MCC}$. 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 qnorm plot 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].

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.

8. IR is defined as the ratio of the number of the majority class instances to the number of the minority class instances [67].
TABLE 7: Description of the 27 Data Sets

<table>
<thead>
<tr>
<th>Data</th>
<th>Modules</th>
<th>Defective Modules</th>
<th>IR</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>ant-1.3</td>
<td>125</td>
<td>20</td>
<td>5.25</td>
<td>[68], [69]</td>
</tr>
<tr>
<td>ant-1.4</td>
<td>178</td>
<td>40</td>
<td>3.45</td>
<td>[68], [69]</td>
</tr>
<tr>
<td>ant-1.5</td>
<td>293</td>
<td>32</td>
<td>8.16</td>
<td>[68], [69]</td>
</tr>
<tr>
<td>ant-1.6</td>
<td>351</td>
<td>92</td>
<td>2.82</td>
<td>[68], [69]</td>
</tr>
<tr>
<td>camel-1.0</td>
<td>339</td>
<td>13</td>
<td>25.08</td>
<td>[68], [69]</td>
</tr>
<tr>
<td>camel-1.2</td>
<td>608</td>
<td>216</td>
<td>1.81</td>
<td>[68], [69]</td>
</tr>
<tr>
<td>camel-1.4</td>
<td>872</td>
<td>145</td>
<td>5.01</td>
<td>[68], [73], [69]</td>
</tr>
<tr>
<td>camel-1.6</td>
<td>965</td>
<td>188</td>
<td>4.13</td>
<td>[68], [73], [69], [70], [71]</td>
</tr>
<tr>
<td>ivy-2.0</td>
<td>352</td>
<td>40</td>
<td>7.8</td>
<td>[68], [72], [69]</td>
</tr>
<tr>
<td>jedit-3.2</td>
<td>272</td>
<td>90</td>
<td>2.02</td>
<td>[68], [69], [76], [77]</td>
</tr>
<tr>
<td>jedit-4.0</td>
<td>306</td>
<td>75</td>
<td>3.08</td>
<td>[68], [69], [76], [77], [70]</td>
</tr>
<tr>
<td>jedit-4.1</td>
<td>312</td>
<td>79</td>
<td>2.95</td>
<td>[68], [76], [77]</td>
</tr>
<tr>
<td>jedit-4.3</td>
<td>367</td>
<td>48</td>
<td>6.65</td>
<td>[68], [76], [77]</td>
</tr>
<tr>
<td>jedit-4.3</td>
<td>492</td>
<td>11</td>
<td>43.73</td>
<td>[68], [72], [76], [77]</td>
</tr>
</tbody>
</table>

**TABLE 8: Summary statistics for predictive performance (MCC)**

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>-0.112</td>
<td>Max</td>
<td>0.679</td>
</tr>
<tr>
<td>Mean</td>
<td>0.319</td>
<td>Median</td>
<td>0.335</td>
</tr>
<tr>
<td>sd</td>
<td>0.137</td>
<td>Trimmed (0.2) sd</td>
<td>0.140</td>
</tr>
<tr>
<td>Skewness</td>
<td>-0.271</td>
<td>Kurtosis</td>
<td>2.672</td>
</tr>
</tbody>
</table>

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.

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

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>-0.258</td>
<td>Max</td>
<td>0.504</td>
</tr>
<tr>
<td>Mean</td>
<td>0.050</td>
<td>Trimmed mean</td>
<td>0.033</td>
</tr>
<tr>
<td>sd</td>
<td>0.092</td>
<td>Trimmed (0.2) sd</td>
<td>0.069</td>
</tr>
<tr>
<td>Skewness</td>
<td>1.246</td>
<td>Kurtosis</td>
<td>5.041</td>
</tr>
</tbody>
</table>
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.

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

<table>
<thead>
<tr>
<th>IR</th>
<th>Average Improvement</th>
<th>Cliff’s δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1~2</td>
<td>0.013 (0.011, 0.014)</td>
<td>+S (+S +S)</td>
</tr>
<tr>
<td>2~4</td>
<td>0.022 (0.020, 0.023)</td>
<td>+M (+S +M)</td>
</tr>
<tr>
<td>4~8</td>
<td>0.048 (0.046, 0.050)</td>
<td>+L (+L +L)</td>
</tr>
<tr>
<td>8~16</td>
<td>0.053 (0.048, 0.058)</td>
<td>+M (+M +M)</td>
</tr>
<tr>
<td>16~64</td>
<td>0.076 (0.070, 0.082)</td>
<td>+L (+M +L)</td>
</tr>
</tbody>
</table>

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

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.

![Boxplot of differences in performance (MCC) with and without imbalanced learning by classifiers](image1)

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

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Average Improvement</th>
<th>Cliff’s δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>0.145 (0.140, 0.151)</td>
<td>+L (+L +L)</td>
</tr>
<tr>
<td>C4.5</td>
<td>0.042 (0.040, 0.045)</td>
<td>+M (+M +L)</td>
</tr>
<tr>
<td>LR</td>
<td>0.038 (0.036, 0.040)</td>
<td>+L (+L +L)</td>
</tr>
<tr>
<td>Ripper</td>
<td>0.028 (0.026, 0.031)</td>
<td>+S (+S +M)</td>
</tr>
<tr>
<td>IBk</td>
<td>0.023 (0.021, 0.025)</td>
<td>+S (+S +M)</td>
</tr>
<tr>
<td>RF</td>
<td>0.018 (0.017, 0.020)</td>
<td>+M (+S +M)</td>
</tr>
<tr>
<td>NB</td>
<td>0.006 (0.004, 0.007)</td>
<td>+N (+N +S)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Input Metrics</th>
<th>Average Improvement</th>
<th>Cliff’s δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK</td>
<td>0.043 (0.039, 0.046)</td>
<td>+M (+M +L)</td>
</tr>
<tr>
<td>NET</td>
<td>0.035 (0.033, 0.038)</td>
<td>+M (+M +M)</td>
</tr>
<tr>
<td>PROC</td>
<td>0.036 (0.033, 0.039)</td>
<td>+M (+M +M)</td>
</tr>
<tr>
<td>CK+NET</td>
<td>0.035 (0.033, 0.038)</td>
<td>+M (+M +L)</td>
</tr>
<tr>
<td>CK+PROC</td>
<td>0.027 (0.024, 0.030)</td>
<td>+M (+M +M)</td>
</tr>
<tr>
<td>NET+PROC</td>
<td>0.030 (0.027, 0.032)</td>
<td>+M (+M +M)</td>
</tr>
<tr>
<td>CK+NET+PROC</td>
<td>0.029 (0.027 0.032)</td>
<td>+M (+M +M)</td>
</tr>
</tbody>
</table>

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.

![Boxplot of differences in performance (MCC) with and without imbalanced learning by metrics](image2)

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).
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 $\delta$ 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. This is indicated by the graying out of the non-significant 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.

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

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Improvement</th>
<th>Cliff’s $\delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>UOBag</td>
<td>0.069 (0.065, 0.074)</td>
<td>+L (+L,+L)</td>
</tr>
<tr>
<td>UBag</td>
<td>0.067 (0.063, 0.072)</td>
<td>+L (+L,+L)</td>
</tr>
<tr>
<td>SBag</td>
<td>0.058 (0.054, 0.062)</td>
<td>+L (+L,+L)</td>
</tr>
<tr>
<td>OBag</td>
<td>0.049 (0.045, 0.054)</td>
<td>+L (+L,+L)</td>
</tr>
<tr>
<td>EM1v1</td>
<td>0.047 (0.043, 0.052)</td>
<td>+L (+L,+L)</td>
</tr>
<tr>
<td>Ubst</td>
<td>0.039 (0.035, 0.044)</td>
<td>+M (+M,+L)</td>
</tr>
<tr>
<td>SMOTE</td>
<td>0.034 (0.030, 0.039)</td>
<td>+M (+M,+M)</td>
</tr>
<tr>
<td>COS</td>
<td>0.030 (0.027, 0.033)</td>
<td>+L (+M,+L)</td>
</tr>
<tr>
<td>Sbst</td>
<td>0.030 (0.026, 0.034)</td>
<td>+M (+S,+M)</td>
</tr>
<tr>
<td>OS</td>
<td>0.023 (0.019, 0.027)</td>
<td>+S (+S,+S)</td>
</tr>
<tr>
<td>Obst</td>
<td>0.022 (0.018, 0.026)</td>
<td>+S (+S,+S)</td>
</tr>
<tr>
<td>UObst</td>
<td>0.022 (0.018, 0.026)</td>
<td>+S (+S,+S)</td>
</tr>
<tr>
<td>US</td>
<td>0.017 (0.012, 0.021)</td>
<td>+N (+N,+S)</td>
</tr>
<tr>
<td>Bag</td>
<td>0.013 (0.011, 0.015)</td>
<td>+M (+S,+M)</td>
</tr>
<tr>
<td>UOS</td>
<td>0.013 (0.008, 0.018)</td>
<td>+N (+N,+N)</td>
</tr>
<tr>
<td>Bst</td>
<td>0.008 (0.005, 0.011)</td>
<td>+N (+N,+S)</td>
</tr>
</tbody>
</table>

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.
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, UBOBag, 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.

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 available11. 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, kNN, 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 mitigate 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
**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 neighbors that have a relationship represented by an edge with the “ego” node. This describes how a node is connected to its neighbours, for example, in Fig. 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., \( \text{Size} \times (\text{Size} - 1) \).
- **Density of ego network (Density)** is the percentage of possible ties that are actually present, i.e., \( \frac{Tie}{Pairs} \).
- **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., \( \frac{\text{WeakComp}}{\text{Size}} \).
- **TwoStepReach** is the percentage of nodes that are two steps away.
- **The reach efficiency (ReachEfficiency)** normalizes TwoStepReach by size, i.e., \( \frac{\text{TwoStepReach}}{\text{Size}} \).
- **Brokage** 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.
- **nBrokage** is the Brokage normalized by the number of pairs, i.e., \( \frac{\text{Brokage}}{\text{Pairs}} \).
- **EgoBetween** is the percentage of shortest paths 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 (EjffSize)** 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 to 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.
- **nEigvector** 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:
- **REVISINGs** 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 the 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**


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.