Predicting Faults in High Assurance Software

Naeem Seliya*
Taghi M. Khoshgoftaar†
Jason Van Hulse‡

Abstract

Reducing the number of latent software defects is a development goal that is particularly applicable to high assurance software systems. For such systems, the software measurement and defect data is highly skewed toward the not-fault-prone program modules, i.e., the number of fault-prone modules is relatively very small. The skewed data problem, also known as class imbalance, poses a unique challenge when training a software quality estimation model. However, practitioners and researchers often build defect prediction models without regard to the skewed data problem. In high assurance systems, the class imbalance problem must be addressed when building defect predictors. This study investigates the roughly balanced bagging (RBBag) algorithm for building software quality models with data sets that suffer from class imbalance. The algorithm combines bagging and data sampling into one technique. A case study of 15 software measurement data sets from different real-world high assurance systems is used in our investigation of the RBBag algorithm. Two commonly used classification algorithms in the software engineering domain, Naïve Bayes and C4.5 decision tree, are combined with RBBag for building the software quality models. The results demonstrate that defect prediction models based on the RBBag algorithm significantly outperform models built without any bagging or data sampling. The RBBag algorithm provides the analyst with a tool for effectively addressing class imbalance when training defect predictors during high assurance software development.

Keywords: *defect prediction; software measurements; imbalanced data; bagging; data sampling; classification.*

*Naeem Seliya is with the Computer and Information Science Department, University of Michigan–Dearborn, Dearborn, Michigan; nseliya@umich.edu.

†Taghi M. Khoshgoftaar is with the Computer and Electrical Engineering and Computer Science Department, Florida Atlantic University, Boca Raton, Florida; khoshgof@fau.edu.

‡Jason Van Hulse is with First Data, Inc., Coral Springs, Florida; jvanhulse@gmail.com.

1 Introduction

Software quality assurance is important during the development of high assurance software systems. Various techniques, such as unit testing, inspections, technical reviews, system testing, and defect prediction are employed to ensure a very low number of latent defects in high assurance systems. Software quality modeling and estimation (also referred to as defect prediction) is a widely researched area in the software engineering domain [14, 20, 22, 23, 17, 26, 34].

A software defect prediction model is typically trained with software measurement and defect data from prior development experiences, and the validated model is then applied to estimate the quality, e.g., fault-prone (*fp*) or not-fault-prone (*nfp*), of the target program modules. This allows the project management team to intelligently allocate project resources toward program modules that are more likely to be of poor quality.

In the literature, one can find works where researchers and practitioners build defect prediction models by simply training a model on the available software measurement and defect data [14, 20, 23]. More specifically, little or no attention is given to the characteristics of the underlying data used for modeling. A typical high assurance system is developed using processes that aim to effectively reduce the number of software defects, i.e., the proportion of fault-prone program modules is relatively very small compared to the proportion of not-fault-prone modules. Consequently, software measurement and defect data of such systems will be highly skewed in the favor of the *nfp* program modules. This problem, also known as class imbalance, poses a unique challenge when building software quality estimation models.

A defect predictor trained on a skewed software measurement data set will be impractical since the model will be affected by the overrepresentation of the *nfp* instances (program modules), yielding a model that is poor in efficiently predicting the *fp* instances. Thus, it is very important to address the class imbalance problem when building defect prediction models for high assurance systems.

One common method to improve defect predictor performance is bagging [3]. This technique uses an ensemble of models, each trained on a randomly sampled subset of the original data set. The ensemble of learners then collectively predicts the class label (e.g., *fp* or *nfp*) of a target instance. Data sampling is a commonly used strategy for working with skewed training data sets [33]. Majority undersampling and minority oversampling are two broad categories of data sampling techniques; majority undersampling removes instances from the larger (negative) class while minority oversampling adds instances to the smaller (positive) class [33]. Roughly Balanced Bagging (RBBag), proposed by Hido and Kashima [13], combines bagging and data sampling to improve the performance of bagged classifiers when learning from imbalanced data.

This study investigates the RBBag algorithm for building software quality models with a skewed software measurement data set. To our knowledge, in software defect prediction this is the first study to investigate combining bagging and data sampling techniques.

An empirical case study of 15 software measurement data sets obtained from nine different high assurance systems is used in our investigation of the RBBag algorithm for defect prediction. Two very commonly used classification algorithms in the software engineering domain, Naïve Bayes and C4.5 decision tree, are investigated for building the software quality models. The software quality models built with the RBBag algorithm are compared with those built without any bagging or data sampling. Key conclusions based on our empirical results are: (1) defect prediction models based on the RBBag algorithm are significantly better than those built without any bagging or data sampling, and (2) the Naïve Bayes classification algorithm outperforms the C4.5 decision tree algorithm, but when combined with RBBag, C4.5 outperforms Naïve Bayes.

The remainder of this paper is organized as follows: Section 2 summarizes some literature relevant to this paper. Section 3 details the Roughly Balanced Bagging algorithm we implemented for our study. Section 4 presents the case study settings, including software measurement data, classification algorithms, performance metrics, and modeling experiments. Section 5 presents and discusses the case study results, including statistical validation. Finally, in Section 6 we summarize this paper and provide suggestions for future work.

2 Related Work

Data sampling is a commonly used strategy for addressing the class imbalance problem [1, 6, 15, 33, 35]. It involves re-adjusting the relative distributions of the minority and majority classes in the training data set. Majority undersampling and minority oversampling are the two primary types of data sampling techniques for handling class imbalance. We have evaluated data sampling techniques for software reliability modeling in our previous work [32, 29].

Majority undersampling works by removing a given number of instances from the majority class (*nfp* modules in our study). Some majority undersampling techniques include random undersampling [33], one-sided selection [19], and Wilson's editing [1]. Minority oversampling works by adding a given number of instances to the minority class (*fp* modules in our study). Some minority oversampling techniques include random oversampling [33], cluster-based oversampling [15], Synthetic Minority Oversampling Technique (SMOTE) [4], and Borderline SMOTE [12].

Bagging is a meta-learning technique that builds an ensemble of learners for improving classification performance [3]. In order to improve learning with imbalanced data, bagging has been combined with data sampling. Exactly Balanced Bagging (EBBag) combines all the minority class instances with random subsets of the majority class instances, such that the relative class distributions is balanced when each model in the ensemble is built [21, 25]. Roughly Balanced Bagging [3], while similar to EBBag, relates more closely to the original bagging algorithm by allowing for changes in the training data set class distributions. RBBag combines bagging with the random undersampling technique [3].

In this study, we examine the RBBag algorithm for building defect predictors with imbalanced data sets. Our preliminary investigation indicated that RBBag and EBBag performed relatively similar in terms of classification model performance [18]. Hence, we only study RBBag in this paper.

A comparative study of other techniques, such as boosting [30, 5], for addressing class imbalance in software defect prediction is out of scope for this paper. However, our future work will include a comparison of boosting and bagging methods for software defect prediction with skewed data.

3 Roughly Balanced Bagging

Bagging builds an ensemble of base learners using sampled subsets of the training data, and then aggregates their estimations to obtain the final prediction [3]. The training data is denoted by $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where x_i is the input feature vector of d-dimensional real-valued or nominal-valued variables, and y_i is the class label of x_i. In bagging, the training data D is converted into K equal-sized subsets $\{D^1, D^2, \ldots, D^K\}$ using bootstrap sampling. In our study, K is set to 10 which is commonly done. If $f^k(x)$ is the base model trained on the k-th subset D^k, and $f^A(x)$ is the final ensemble model, then the output of $f^A(x)$ is aggregated from the set of base models $\{f^1(x), f^2(x), \ldots, f^K(x)\}$. Hido and Kashima [13] state that though the original bagging algorithm [3] determines $f^A(s)$ through a voting of the predicted class labels, they use the average of the estimated probability $p^k(y|x)$ as shown below.

$$f^A(x_i) = \frac{1}{K}\sum_{k=1}^{K} p^k(y_i|x_i) \qquad (1)$$

The average of the estimated probability is used instead of a majority voting based on a recommendation by Fan et al. [8], that the former provides better prediction accuracies.

The relative distributions of the positive and negative instances in the sampled subsets need to be corrected in order to build learners that perform satisfactorily for both classes. The algorithm for the Roughly Balanced Bagging (RBBag) technique, proposed by Hido and Kashima [13], is shown in Figure 1. The key aspect of the algorithm is determining the numbers of instances of the two (positive and negative) classes to form the subsets. The number of positive (minority) instances is set to that in the original data set. If the instances are sampled without replacement, then all of the positive instances will be included in all of the sampled subsets. The number of negative instances to be sampled is determined probabilistically using the negative binomial distribution, which is based on two parameters – the number of minority instances (r) to be sampled and the probability of success $q = 0.5$. When r is an integer, it's also referred as the Pascal distribution. Instances of both classes are sampled with equal probability, where the size of the positive class is kept constant and the size of the negative (majority) class varies.

For each iteration k of RBBag, an independent training data set is sampled from the original data. Initially, N_{c_1} minority instances are added, where N_{c_1} is the number of minority class instances in the original data. Subsequently, a random number, w from the Pascal distribution with parameters $r = N_{c_1}$ and q is generated, and w majority instances are added to that iteration's training dataset. If sampling is done without replacement, then w must be reduced to N_{c_0} (number of majority instances in the original data set) if it is too large. On the other hand, when sampling is done with replacement, then w does not need to be limited. Once the training data set for each iteration is created, a model is trained on that data set. The above process of building the training data set and training a model on it is repeated for all K iterations. Once the final ensemble is formed, a new instance is evaluated by all k models and the average of the posterior probabilities they produce is determined as the final output.

We implemented the RBBag algorithm within the Weka data mining framework [36]. The algorithm was implemented for both sampling with replacement and sampling without replacement. In this study we present results of the sampling without replacement version of RBBag. Friedman and Hall [10] indicate that there is no significant difference between the results obtained by sampling with replacement and sampling without replacement. In addition, our previous work found RBBag without replacement outperformed RBBag with replacement [18]. Compared to Hido and Kashima [13], our implementation of RBBag allows changing the probability of sampling a positive instance, i.e., q. In this study, we present results when $q = 0.5$. Future work will investigate other settings for q when building defect prediction models; in particular, a study on how the balance in the data set is maintained with respect to different values of q.

Figure 1. Roughly Balanced Bagging

```
Inputs:
    $D$ is the training data set
    $L$ is a the base learner
    $K$ is the number of base learners
    $x_i$ is an instance drawn from the test set

Build Roughly Balanced Bagging Model ($D$,$L$,$K$):
    Divide $D$ into negative set $D^{neg}$
        and positive set $D^{pos}$
    For $k$ = 1 to $K$
        Draw $N_k^{neg}$ from the negative binomial
            distribution with $n = N_k^{pos}$ and $q = 0.5$.
        Set $N_k^{pos}$ as the size of $D^{pos}$ (i.e., $|D^{pos}|$)
        Let $D_k^{neg}$ be $N_k^{neg}$ instances sampled
            from $D^{neg}$ with or without replacement
        Let $D_k^{pos}$ be $N_k^{pos}$ instances sampled
            from $D^{pos}$ with or without replacement
        Build a model $f^k(x)$ by applying $L$ to
            $D_k^{neg}$ and $D_k^{pos}$
    Combine all $f^k(x)$ into the aggregated model $f^A(x)$
    Return $f^A(x)$

Predict ($f^A(x)$, $x_i$, y):
    Calculate $p^A(y|x_i) = \frac{1}{K}\sum_{k=1}^{K} p^k(y|x_i)$ for all $y$
    Let $\hat{y} = \arg\max p^A(y|x_i)$
    Return $\hat{y}$
```

4 Empirical Settings

4.1 Software Measurement and Defect Data

The 15 software measurement and defect data sets were obtained from nine real-world software systems [17, 31, 16]. Important information for those data sets is provided in Table 1, including number of software metrics, number of fault-prone program modules, number of not-fault-prone modules, total number of modules, and the proportion of *fp* modules in the dataset. The use of specific software metrics was governed primarily based on their availability for analysis purposes. A different project may consider a different set of software measurements for analysis [7, 9, 11, 24].

The last column in Table 1 provides insight into the varying degrees of skewed data that are considered in our study. The highest and lowest proportions of *fp* modules in a given training data set are 29.43% (CCCS-2) and 1.30% (SP3), respectively. The selection of a wide ranges of data skewness makes the results of our work more widely acceptable, including many commercial and industrial-strength software systems. We also selected a wide range of dataset sizes, with 282 modules being the smallest (CCCS) and 8850 modules being the largest (JM1). The selection of a wide range of the degree of class imbalance and the data set sizes provides the analyst with a better insight into building defect prediction models for high assurance systems. The following list summarizes the different high-assurance software systems and their software measurement data sets used in our case studies.

1. The SP1, SP2, SP3, and SP4 data sets represent four successive releases of a large legacy telecommunications system, and the case study data was based on 42 software metrics which included 24 product metrics, 14 process metrics, and four execution metrics [17]. The software system is an embedded-computer application that included finite-state machines. Using the procedural development paradigm, the software was written in PROTEL (a high-level programming language) and was maintained by professional programmers in a large organization. Fault data was collected at the module-level by the problem reporting system. A module was considered as *nfp* if it had no post-release faults, and *fp* otherwise. The number of program modules in the four data sets are: 3649 for SP1, 3981 for SP2, 3541 for SP3, and 3978 for SP4. The relative distribution of the *fp* and *nfp* modules for SP1, SP2, SP3, and SP4 are shown in Table 1.

2. The JM1 project, written in C, is large project for a real-time ground system that uses simulations to generate predictions for missions. The JM1 dataset consisted of 10,883 program modules, of which 2,105 modules had software defects (ranging from 1 to 26) while the remaining $8,778$ modules were defect-free. The dataset contained some inconsistent modules, i.e., those with identical software measurements but with different class labels. Upon removing such modules the dataset was reduced to $8,850$ modules, consisting of 1,687 *fp* modules (i.e., with one or more defects) and 7,163 *nfp* modules (i.e., with no defects). This number-of-defects-based definition of *fp* and *nfp* program modules also applies to the CM1, KC1, KC2, KC3, MW1, and PC1 data sets.

   Each program module in the JM1 data sets was characterized by 13 basic software product metrics [31]. These same metrics were also used for the CM1, KC1, KC2, KC3, MW1, and PC1 data sets, and they were primarily governed by their availability, internal workings of the projects, and the data collection tools used. The type and numbers of metrics made available were solely determined by the NASA Metrics Data Program. Other metrics, including software process and object-oriented metrics, were not available at the time of modeling and analysis.

3. The KC1 project is a single CSCI (Computer Software Configuration Item) within a large ground system and consists of 43 KLOC (thousand lines of code) in C++. A given CSCI consists of logical groups of computer software components (CSCs). The dataset contains 2107 modules, of which 325 are *fp* and 1782 are *nfp*. The maximum number of faults in a program module is 7.

4. The KC2 project, written in C++, is the science data processing unit of a storage management system used for receiving and processing ground data for missions. The dataset includes only those modules that were developed by NASA software developers and not COTS (Commercial-Off-The-Shelf) software. The dataset contains 520 modules, of which 106 are *fp* and 414 are *nfp*. The maximum number of faults in a software module is 13.

5. The KC3 project, written in 18 KLOC in Java, is a software application that collects, processes, and delivers satellite meta-data. The dataset contains 458 modules, of which 43 are *fp* and 415 are *nfp*. The maximum number of faults in a module is 6.

6. The CM1 software measurement dataset is that of a science instrument application written in C with approximately 20 KLOC. The dataset contains 505 modules, of which 48 are *fp* and 457 are *nfp*. The maximum number of faults in a module is 5.

7. The MW1 project is a software application from a zero gravity combustion experiment that has been completed. The MW1 dataset consists of 8000 lines in C. The dataset contains 403 modules, of which 31 are *fp* and 372 are *nfp*. The maximum number of faults in a module is 4.

8. The PC1 project is flight software from an earth orbiting satellite that is no longer operational. It consists of 40 KLOC in C. The software measurement dataset contains 1107 modules, of which 76 are *fp* and 1031 are *nfp*. The maximum number of faults in a module is 9.

Table 1. Software Measurement Data Sets

Data set	# metrics	# *fp*	# *nfp*	# total	% *fp*
SP1	42	230	3419	3649	6.30
SP2	42	189	3972	3981	4.75
SP3	42	46	3495	3541	1.30
SP4	42	92	3886	3978	2.31
CCCS-2	8	83	199	282	29.43
CCCS-4	8	55	227	282	19.50
CCCS-8	8	27	255	282	9.57
CCCS-12	8	16	266	282	5.67
CM1	13	48	457	505	9.50
JM1	13	1687	7163	8850	19.06
KC1	13	325	1782	2107	15.42
KC2	13	106	414	520	20.38
KC3	13	43	415	458	9.39
MW1	13	31	372	403	7.69
PC1	13	76	1031	1107	6.87

9. The CCCS-2, CCCS-4, CCCS-8, and CCCS-12 data sets represent software metrics and defect data for a large military command, control, and communications system (CCCS) [16]. The CCCS data sets are based on software product metrics. The numerical suffix for the CCCS project represents the number of defects threshold used for determining whether a program module is considered *fp* or *nfp*.

4.2 Classification Algorithms

The C4.5 decision tree algorithm is one of the two learners used to build software defect prediction models in this study. We used the version of the learner implemented in Weka [36]. C4.5 is the benchmark decision tree learning algorithm proposed by Quinlan [28], and is one of the most commonly used learners in software engineering research [17, 20]. The decision tree is built using an entropy-based splitting criterion stemming from information theory. We use Weka's default parameters for C4.5 in our experiments.

The Naïve Bayes (NB) learner is based on the Bayesian rule of conditional probability, and assumes that predictor attributes are independent of each other given the class. Naïve Bayes is another very commonly used learner in the software engineering domain, and has been shown to perform well for defect prediction [20, 23]. We use Weka's default parameters for NB in our experiments.

For additional details on the two classification algorithms and their modeling parameters the reader is referred to [36].

4.3 Performance Metrics

A binary (positive and negative) classification problem (such as *fp* and *nfp*) has a confusion matrix consisting of four cells, i.e., true positive (#TP), false positive (#FP), true negative (#TN), and false negative (#FN). If the positive class represents *fp* modules and the negative class represents *nfp* modules, then a false positive indicates an error in which a *nfp* program module is incorrectly classified as *fp*. Similarly, a false negative indicates an error in which a *fp* program module is incorrectly classified as *nfp*. A false positive leads to wasted resources due to inspection of an already good quality program module, whereas a false negative is the more serious error type as it represents a lost opportunity to detect an actual *fp* module.

The remainder of this section provides a brief overview of the classifier performance metrics used in this study. The four values of the two-by-two confusion matrix are used to compute the four basic accuracy/error rates: TPR is the true positive rate, FPR is the false positive rate, TNR is the true negative rate, and FNR is the false negative rate. These are computed as follows:

$$TPR = \frac{\#TP}{N_{c_1}} \quad (2)$$

$$TNR = \frac{\#TN}{N_{c_0}} \quad (3)$$

$$FPR = \frac{\#FP}{N_{c_0}} \quad (4)$$

$$FNR = \frac{\#FN}{N_{c_1}} \quad (5)$$

where N_{c_1} is the total number of positive instances, i.e., *fp* modules, and N_{c_0} is the total number of negative instances, i.e., *nfp* modules. The overall accuracy (ACR) is given by,

$$ACR = \frac{\#TP + \#TN}{N_{c_1} + N_{c_0}} \quad (6)$$

The F-Measure (FM) is based on two information retrieval metrics, Recall (or Effectiveness) and Precision (or Efficiency), where Recall is the TPR and Precision is the ratio of the number of true positives to the total of the number of true positives and the number of false positives [36]. The default F-Measure is then derived from these two metrics and is given by,

$$FM = \frac{2 \times Recall \times Precision}{Recall + Precision} \quad (7)$$

The Geometric Mean (GM) is another useful singular performance metric for classification problems involving training with skewed data [36]. A higher GM value indicates that a classifier is balanced (not biased to any one class) and shows good performance for both classes. The default Geometric Mean is computed as,

$$GM = \sqrt{(TPR) \cdot (TNR)} \quad (8)$$

An ROC (Receiver's Operating Characteristic) curve is a visual representation of a classifier's performance in which the model's TPR (y-axis) is plotted against its FPR rate (x-axis) [27]. A desirable ROC curve is one that maximizes the Area under the ROC Curve (AROC). Since an ROC curve does not show bias towards the majority class, the AROC value serves as a good singular performance metric when dealing with skewed data. The AROC provides a rating for the classifier's general predictive ability regardless of class prior-probabilities and misclassification costs [36].

The Area under the Precision-Recall Curve (APRC), originating from the information retrieval domain, is another singular performance metric. The APRC metric emphasizes a trade-off between Recall and Precision, where Precision is plotted on the x-axis while Recall is plotted on the y-axis [36]. Similar to AROC, its value ranges from 0 to 1. A desired classifier in terms of APRC is one that maximizes the APRC.

Note that TPR, TNR, FNR, FPR, ACR, FM, and GM utilize the default decision threshold of 0.5 when determining the predicted class of a test example. ROC curves and Precision-Recall curves, however, vary the decision threshold when classifying test examples, and plot the false positive-true positives rates in the first case and precision-recall rates in the second case, as the decision threshold varies.

4.4 Modeling Experiments

For a given learner (NB and C4.5) 10-fold cross validation is used to train the defect prediction models, where nine folds are used as training data and the tenth fold is used as test data. The cross validation process is repeated 10 times to avoid any bias due to a lucky/unlucky split when forming the data folds. Thus, for a given learner (C4.5 and NB) and a given technique (RBBag and None), 100 models were built for each of the 15 data sets. A total of 6000 defect prediction models were built in this study. The average results of the 10 runs of 10-fold cross validation over the 15 data sets are reported.

5 Results and Analysis

The primary focus of this study is to investigate the performances of the RBBag-associated defect predictors with those built without any bagging. However, for completeness sake a comparison between the two classification algorithms is also presented. The performance of the defect prediction models with respect to the performance metrics discussed earlier are summarized in Table 2. For a given modeling technique and a given learner, the table shows the following performance metrics: TPR, TNR, FPR, FNR, ACR, AROC, APRC, FM, and GM. The problem associated with class imbalance can be seen in the None models (i.e., without RBBag), where the FNR is high for both learners. This implies that many of the program modules predicted as *nfp* are actually *fp*. Thus, despite having a high ACR value, the None models are not practical. The RBBag models have lower ACR values than the None models, but their FNR and TPR values are much better. Such models are of more use to the practitioner than the None models shown in the table.

Table 2. Defect Prediction Results

Performance	None		RBBag	
Metric	NB	C4.5	NB	C4.5
TPR	0.4690	0.3181	0.5238	0.8099
TNR	0.9220	0.9567	0.8957	0.7349
FPR	0.0780	0.0433	0.1043	0.2651
FNR	0.5310	0.6819	0.4762	0.1901
ACR	0.8716	0.9009	0.8519	0.7449
AROC	0.8205	0.6700	0.8218	0.8436
APRC	0.4376	0.3174	0.4459	0.4573
FM	0.3921	0.3349	0.3820	0.3809
GM	62.82	44.54	65.94	76.51

The RBBag models provide better performances than the None models in terms of the AROC, APRC, FM, and GM performance metrics. This is true for both learners, NB and C4.5 (with the sole exception of FM with the NB learner). A closer look at competing models built with a given learner reveals that the C4.5 models benefit more with the RBBag algorithm as compared to the NB models. This may be a characteristic of the C4.5 algorithm that is worth studying in future works.

In order to gauge the significance of the differences between the competing defect prediction models we perform statistical analysis based on the AROC performance metric. A two-way ANOVA (Analysis of Variance) experiment is performed [2], where Factor A represents the two modeling techniques, RBBag and None, and Factor B represents the two learners, NB and C4.5. The null hypothesis is that the average AROC performance of each level of the two factors is similar, while the alternate hypothesis is that at least one level of the two factors is dissimilar. The ANOVA experiment was performed only after satisfying the underlying assumptions of an ANOVA study.

The result of the ANOVA experiment is summarized in Table 3, which shows the degrees of freedom, sum of squares, mean square, F-statistic, and p-value for each of the two factors. The p-values allows us to reject the null hy-

Table 3. ANOVA Test Results

Source	df	SS	MS	F	Pr > F
Learner	1	6.21	6.21	368.37	< 0.0001
Technique	1	11.48	11.48	680.39	< 0.0001
Error	5997	101.17	0.017		
Total	5999	118.86			

Table 4. Multiple Comparison Results for Techniques

Mean	Learner	Tukey's Grouping
0.8327	RBBag	A
0.7453	None	B

pothesis, and accept the alternate hypothesis as stated earlier. The ANOVA experiment was performed at the significance level of $\alpha = 0.05$.

Multiple pairwise comparisons reveal if two populations are indeed significantly different from each other. We perform multiple pairwise comparisons using Tukey's Honestly Significant Difference (HSD) Test [2]. These were performed to respectively compare the two modeling techniques (RBBag and None) and the two learners (C4.5 and NB). The AROC performance metrics is used as the response variable for the multiple pairwise comparisons. The multiple pairwise comparisons were performed at the significance level of $\alpha = 0.05$.

The significance-based groupings obtained from the multiple pairwise comparisons based on Tukey's HSD Test for the two modeling techniques are shown in Table 4. Two levels of a factor show no significant difference if they have the same letter in common. The RBBag models are significantly better than the None models. This validates our claim that the class imbalance problem must be addressed when building defect prediction models for high assurance software systems. Ignoring the problem will yield performances similar to the None models, which are clearly inferior defect prediction models.

The results of the multiple pairwise comparisons for the two learners are shown in Table 5, which once again shows

Table 5. Multiple Comparison Results for Learners

Mean	Learner	Tukey's Grouping
0.8212	NB	A
0.7568	C4.5	B

the groupings based on Tukey's HSD Test. The NB learner is significantly better than the C4.5 learner, in the context of this study (averaged over both scenarios with and without the use of RBBag). Other researchers have also demonstrated the good performance of the Naïve Bayes algorithm for software defect prediction [14, 23]. From Table 2, however, the improvement in NB with RBBag is modest, while for C4.5, the improvement is dramatic. Without RBBag, NB substantially outperforms C4.5 (relative to AROC, APRC, FM, and GM) but when combined with RBBag, C4.5 outperforms NB relative to these metrics, with the exception of FM. Clearly when data is imbalanced, it is highly recommended to use C4.5 with RBBag.

5.1 Threats to Validity

An important component of experimental work in the domain of software defect prediction is a discussion on threats to validity [37], which are categorized into threats to $internal$ validity and threats to $external$ validity.

Threats to internal validity relate to unaccounted influences that may impact the experimental results. Our research group implemented the RBBag algorithm within the Weka framework, and carefully tested the code to make sure that it worked properly. The NB and C4.5 algorithms are already available within Weka, and they were used without modification in these experiments. Different members of the research team analyzed the results to validate the output.

Threats to external validity consider the generalizability of the results outside of the experimental setting, and what limits, if any, should be applied. Our experiments were conducted using 15 different sofware measurement data sets, which makes the study very comprehensive. These datasets vary in size, number of attributes, and level of class imbalance. However, as with any empirical work, experiments with additional datasets would be desirable to confirm our conclusions. Two different learning algorithms were utilized, and they each exhibit different behaviors. The results clearly demonstrate the lift provided by combining C4.5 and RBBag, compared to using the decision tree alone. For NB, however, the benefits of RBBag are more modest. Therefore, the utility of RBBag depends on the underlying learning algorithm, and we must be careful when generalizing the benefits of RBBag to other learning paradigms. No violations of the assumptions for statistical analysis of the results were encountered, so the inferences drawn in this work are valid.

6 Conclusion

Class imbalance (or skewed data) is an important problem when building software defect prediction models for high assurance systems. This implies that the proportion of the fault-prone program modules is very small compared to the proportion of the not-fault-prone program modules. Very often practitioners and researchers neglect addressing this problem when building defect predictors, which leads to models that are not practical for efficiently guiding the management team in targeting the low quality modules.

This paper investigates the Roughly Balanced Bagging algorithm for software defect prediction with imbalanced data. The algorithm combines bagging and data sampling (majority undersampling) for addressing the class imbalance problem. Defect prediction models built using the RBBag algorithm are compared with those built without any bagging or data sampling. Such a comparison is meant to illuminate the necessity of addressing class imbalance during defect prediction modeling. Two commonly used learners for defect prediction, C4.5 and Naïve Bayes, are used in this study.

An empirical case study of 15 software measurement and defect data sets obtained from several real-world high assurance systems is used in this paper. After building 6000 defect prediction models, the key conclusions made in this study are: (1) the RBBag algorithm effectively addresses the class imbalance problem when building defect prediction models, (2) software quality models based on the RBBag algorithm perform significantly better than those built without any bagging or data sampling techniques, particularly when the C4.5 learner is used, and (3) overall, the Naïve Bayes learner performs significantly better than the C4.5 learner, however when combined with RBBag, C4.5 generally outperforms Naïve Bayes.

Future work will investigate other strategies for addressing class imbalance, such as boosting techniques. An evaluation on how the RBBag and None models perform with respect to varying degrees of data skewness will provide a value-added insight to the practitioner. An examination of other learners for building defect predictors in the context of this study will provide the analyst with a broader perspective. In addition, using software measurement data from other high assurance systems will further validate the findings of this paper.

References

[1] R. Barandela, R. M. Valdovinos, J. S. Sánchez, and F. J. Ferri. The imbalanced training sample problem: Under or over sampling? In *Syntatical and Structural Pattern Recognition/Statistical Pattern Recognition*, pages 806–814, 2004.

[2] M. L. Berenson, M. Goldstein, and D. Levine. *Intermediate Statistical Methods and Applications: A Computer Package Approach.* Prentice-Hall, Englewood Cliffs, NJ, 2 edition, 1983.

[3] L. Breiman. Bagging predictors. *Machine Learning*, 26(2):123–140, 1996.

[4] N. V. Chawla, L. O. Hall, K. W. Bowyer, and W. P. Kegelmeyer. SMOTE: Synthetic minority oversampling technique. *Journal of Artificial Intelligence Research*, (16):321–357, 2002.

[5] N. V. Chawla, A. Lazarevic, L. O. Hall, and K. Bowyer. SMOTEBoost: Improving prediction of the minority class in boosting. In *Proceedings of Principles of Knowledge Discovery in Databases*, pages 107–119, 2003.

[6] C. Drummond and R. C. Holte. C4.5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling. In *Workshop on Learning from Imbalanced Data Sets II, International Conference on Machine Learning*, pages 1–8, Washington, DC, August 2003.

[7] M. English, J. Buckley, and T. Cahill. Fine-grained software metrics in practice. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement, ESEM 2007*, pages 295–304, Madrid, Spain, September 2007.

[8] W. Fan, E. Greengrass, J. McCloskey, P. S. Yu, and K. Drummey. Effective estimation of posterior probabilities: Explaining the accuracy of randomized decision tree approaches. In *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM)*, pages 154–161, Houston, TX, November 2005. IEEE Computer Society.

[9] H. D. Frederiksen and L. Mathiassen. A contextual approach to improving software metrics practices. *IEEE Transactions on Engineering Management*, 55(4):602–616, November 2008.

[10] J. H. Friedman and P. Hall. On bagging and nonlinear estimation. *Journal of Statistical Planning and Inference*, 137(3):669–683, 2007.

[11] T. Gyimothy. To use or not to use? the metrics to measure software quality (developers' view). In *Proceeding of the 13th European Conference on Software Maintenance and Reengineering, CSMR '09*, pages 3–4, Kaiserslautern, Germany, March 2009.

[12] H. Han, W. Y. Wang, and B. H. Mao. Borderline-SMOTE: A new over-sampling method in imbalanced data sets learning. In *In International Conference on Intelligent Computing (ICIC'05). Lecture Notes in Computer Science 3644*, pages 878–887. Springer-Verlag, 2005.

[13] S. Hido and H. Kashima. Roughly balanced bagging for imbalanced data. In *Proceedings of 8th SIAM International Conference on Data Mining*, pages 143–152, Atlanta, GA, April 2008. SIAM.

[14] Y. Jiang, J. Lin, B. Cukic, and T. Menzies. Variance analysis in software fault prediction models. In *Proceedings of the 20th IEEE International Symposium on Software Reliability Engineering*, pages 99–108, Bangalore-Mysore, India, Nov. 16-19 2009.

[15] T. Jo and N. Japkowicz. Class imbalances versus small disjuncts. *SIGKDD Explorations*, 6(1):40–49, 2004.

[16] T. M. Khoshgoftaar and E. B. Allen. Logistic regression modeling of software quality. *International Journal of Reliability, Quality and Safety Engineering*, 6(4):303–317, December 1999.

[17] T. M. Khoshgoftaar and N. Seliya. Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Software Engineering Journal*, 9(3):229–257, 2004.

[18] T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano. Comparing boosting and bagging techniques with noisy and imbalanced data. *IEEE Transactions on Systems, Man & Cybernetics: Part A: Systems and Humans*, 2010. In Press.

[19] M. Kubat and S. Matwin. Addressing the curse of imbalanced training sets: One sided selection. In *Proceedings of the 14th International Conference on Machine Learning*, pages 179–186. Morgan Kaufmann, 1997.

[20] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, July-August 2008.

[21] X.-Y. Liu, J. Wu, and Z.-H. Zhou. Exploratory undersampling for class-imbalance learning. In *Proceedings of the Sixth International Conference on Data Mining*, pages 965–969, Hong Kong, China, 2006. IEEE Computer Society.

[22] Y. Liu, T. M. Khoshgoftaar, and N. Seliya. Evolutionary optimization of software quality modeling with multiple repositories. *IEEE Transactions on Software Engineering*, May 2010.

[23] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, Jan. 2007.

[24] M. Mingzhi and J. Yunfei. A coherent object-oriented (oo) software metric framework model: Software engineering. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, volume 2, pages 68–72, Wuhan, Hubei, China, December 2008.

[25] M. Molinara, M. T. Ricamato, and F. Tortorella. Facing imbalanced classes through aggregation of classifiers. In *Proceedings of the 14th International Conference on Image Analysis and Processing*, pages 43–48, Modena, Italy, 2007. IEEE Computer Society.

[26] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on Software Engineering*, 33(6):402–419, June 2007.

[27] F. Provost and T. Fawcett. Robust classification for imprecise environments. *Machine Learning*, 42:203–231, 2001.

[28] J. R. Quinlan. *C4.5: Programs For Machine Learning*. Morgan Kaufmann, San Mateo, California, 1993.

[29] C. Seiffert, T. M. Khoshgoftaar, and J. Van Hulse. Improving software quality predictions with data sampling and boosting. *IEEE Transactions on Systems, Man & Cybernetics: Part A: Systems and Humans*, 39(6):1283–1294, November 2009.

[30] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano. RUSBoost: A hybrid approach to alleviating class imbalance. *IEEE Transactions on Systems, Man & Cybernetics: Part A: Systems and Humans*, 40(1):185–197, January 2010.

[31] N. Seliya and T. M. Khoshgoftaar. Software quality analysis of unlabeled program modules with semi-supervised clustering. *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*, 37(2):201–211, March 2007.

[32] J. Van Hulse, T. M. Khoshgoftaar, and A. Napolitano. Software reliability modeling using skewed measurement data. In *Proceedings of the* 13^{th} *ISSAT International Conference on Reliability and Quality in Design*, pages 181–185, Seattle, WA, August 2007.

[33] J. Van Hulse, T. M. Khoshgoftaar, and A. Napolitano. Experimental perspectives on learning from imbalanced data. In *Proceedings of 24th International Conference on Machine Learning (ICML)*, pages 935–942, June 2007.

[34] Y. Weimin and L. Longshu. A rough set model for software defect prediction. In *Proceedings of the International Conference on Intelligent Computation Technology and Automation (ICICTA)*, pages 747–751, Hunan, China, October 2008.

[35] G. M. Weiss. Mining with rarity: A unifying framework. *SIGKDD Explorations*, 6(1):7–19, 2004.

[36] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, California, 2nd edition, 2005.

[37] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering: An Introduction*. Kluwer International Series in Software Engineering. Kluwer Academic Publishers, Boston, MA, 2000.