An Empirical Study of the Classification Performance of Learners on Imbalanced and Noisy Software Quality Data

Chris Seiffert (chrisseiffert@gmail.com)
Taghi M. Khoshgoftaar (taghi@cse.fau.edu)
Jason Van Hulse (jvanhulse@gmail.com)
Andres Folleco (afolleco@fau.edu)
Florida Atlantic University, Boca Raton, Florida, USA

Abstract

In the domain of software quality classification, data mining techniques are used to construct models (learners) for identifying software modules that are most likely to be fault-prone. The performance of these models, however, can be negatively affected by class imbalance and noise. Data sampling techniques have been proposed to alleviate the problem of class imbalance, but the impact of data quality on these techniques has not been adequately addressed. We examine the combined effects of noise and imbalance on classification performance when seven commonly-used sampling techniques are applied to software quality measurement data. Our results show that some sampling techniques are more robust in the presence of noise than others. Further, sampling techniques are affected by noise differently given different levels of imbalance.

1 Introduction

A common objective in the domain of software quality is to distinguish between program modules (instances) that are *fault-prone (fp)* and *not fault-prone (nfp)*. In doing so, additional resources will be assigned with the objective of detecting and correcting faults in *fp* modules, while resources can be saved by not focusing on modules that are *nfp*. Although data mining techniques have been successfully applied to achieve this goal [7, 9], there are two characteristics common in software quality measurement data that can negatively impact these procedures: imbalance and noise.

In this study, we consider only binary classification, since binary datasets are most prevalent in software quality classification problems. Binary classification data is $imbalanced$ if the prior probabilities of the two classes are not equal. The class with the lower prior probability is called the *minority* class, while the class with the higher prior probability is called the *majority* class. In the domain of software quality, class imbalance can be very severe [8]. For example, an imbalanced software system may consist of 1000 program modules, 950 of which are labeled as nfp and 50 which are labeled fp (the prior probability of the fp class is therefore 5%). High-assurance software systems represent one particular type of system where imbalance in the class label is often present, since program modules have a relatively small likelihood to be fault-prone. Studies have demonstrated that class imbalance can make data mining difficult [17]. *Data sampling* (or resampling) techniques have been proposed in an attempt to alleviate the class imbalance problem.

In the data mining context, noise refers to incorrect data values in a dataset. A special type of noise called *class noise* or *labeling errors*, which occur when noise is located in the class label of an example (program module), is especially harmful to classification performance [21]. Class noise can also have the added effect of contributing to the level of imbalance in a dataset. In software quality, for example, reporting the number of errors in a module is often a task assigned to the software engineer who was responsible for development of that module. Therefore, it is not unlikely that the number of faults in a given software module will be underestimated. This underestimation would cause the number of modules labeled as *fp* (which is already the minority class) to be further reduced, increasing the severity of imbalance within the data.

The objective of this work is to analyze the relationship between data sampling techniques, class imbalance, and class noise in the context of software measurement data. Both class imbalance and class noise (and the interaction between the two) are pervasive issues in empirical software engineering that unfortunately have received little prior attention. Our contributions relate to answering the following research questions:

- How are data sampling techniques affected by the presence of noise in software quality data? What benefit do these techniques provide when applied to a noisy measurement dataset?

1-4244-1500-4/07/$25.00 ©2007 IEEE

- How does the combination of imbalance level and noise level affect these data sampling techniques? Are sampling techniques more affected by noise when imbalance is more severe?

1.1 Related Work

The application of data sampling techniques to alleviate class imbalance has been studied by researchers. Some have compared the performance of undersampling and oversampling techniques [4, 14], while others have evaluated the effect of class distribution on classification performance [18]. Others have proposed and evaluated more "intelligent" methods of sampling [1, 5] designed to add or remove examples from a dataset in a way that benefits the classifier. Additional background on data sampling is in Section 3.

Numerous studies have been performed to analyze and deal with noise. For example, Zhu and Wu study the relative impact of class and attribute noise, and find that class noise has a more significant impact on classification performance [21]. Our research group has studied the impact of noise on software measurement data [10] and proposed numerous solutions for improving data quality [11, 16], including a hybrid cleansing procedure [12] to identify and cleanse noisy instances. We are unaware, however, of any related work that studies the combined effects of both noise and imbalance on sampling technique performance from the perspective of empirical software engineering.

Sections 2 and 3 briefly describe the 11 learners and seven data sampling techniques considered in this study, and Section 4 presents our experimental design. The empirical results are provided in Section 5, and finally conclusions and future work are given in Section 6.

2 Classification Algorithms

We use 11 learners in our experiments, all of which were implemented in the WEKA tool [19], version 3.5.2. Default parameter changes were done only when experimentation showed a general improvement in the classifier performance based on preliminary analysis.

Naive Bayes (NB) utilizes Bayes's rule of conditional probability and is termed 'naive' because it assumes independence of the features. *Logistic regression* (LR) is a statistical regression model for categorical prediction. *RIPPER* (Repeated Incremental Pruning to Produce Error Reduction) is a rule-based learner and is named JRip in WEKA. The *random forests* (RF) classifier [2] uses bagging and the 'random subspace method' to build an ensemble of randomized decision trees which are combined to produce the final prediction. The default WEKA parameters for these four learners were not changed.

C4.5 is a benchmark decision tree learning algorithm. Two different versions of the C4.5 classifier were used. C4.5 (D) uses the default parameter settings in WEKA, while C4.5 (N) uses Laplace smoothing but does not use pruning [18]. For a *Multilayer perceptrons* (MLP) learner (a type of neural network), the 'hiddenLayers' parameter was changed to '3' to define a network with one hidden layer containing three nodes, and the 'validationSetSize' parameter was changed to '10' to cause the classifier to leave 10% of the training data aside to be used as a validation set to determine when to stop the iterative training process. *Radial basis function networks* (RBF) are another type of artificial neural network. The only parameter change for RBF was to set the parameter 'numClusters' to 10. Two *K nearest neighbors* (kNN) learners were built using $k = 2$ and $k = 5$ and were denoted '2NN' and '5NN', respectively. One parameter change was made for kNN: the 'distanceWeighting' parameter was set to 'Weight by 1/distance'. The *support vector machine* (SVM) learner had two changes to the default parameters: the complexity constant 'c' was set to 5.0 and 'buildLogisticModels' was set to 'true'.

3 Sampling Techniques

This section provides a brief overview of the seven sampling techniques considered in this work: random undersampling (RUS), random oversampling (ROS), one-sided selection (OSS), cluster-based oversampling (CBOS), Wilson's editing (WE), SMOTE (SM), and borderline-SMOTE (BSM). For those techniques that require a parameter value to be set, we vary the parameter and consider only the value that yields the best results (these parameters are explained below).

The two most common preprocessing techniques are random *minority oversampling* (ROS) and random *majority undersampling* (RUS). In ROS, instances of the minority class are randomly duplicated in the dataset. In RUS, instances of the majority class are randomly discarded from the dataset.

In one of the earliest attempts to improve upon the performance of random resampling, Kubat and Matwin [13] proposed a technique called *one-sided selection* (OSS). One-sided selection attempts to intelligently undersample the majority class by removing majority class examples that are considered either redundant or 'noisy.'

The technique called *Wilson's editing* [1] (WE) uses the kNN technique with $k = 3$ to classify each example in the training set using all the remaining examples, and removes those majority class examples that are misclassified. Barandela et al. also propose a modified distance calculation, which causes an example to be biased more towards being identified with positive examples than negative ones.

Chawla et al. [3] proposed an intelligent oversampling method called Synthetic Minority Oversampling Technique or SMOTE. SMOTE (SM) adds new, artificial minority examples by extrapolating between preexisting minority instances rather than simply duplicating original examples.

The technique first finds the k nearest neighbors of the minority class for each minority example (the paper recommends $k = 5$). The artificial examples are then generated in the direction of some or all of the nearest neighbors, depending on the amount of oversampling desired.

Han et al. presented a modification of Chawla et al.'s SMOTE technique which they call *borderline-SMOTE* [5] (BSM). BSM selects minority examples which are considered to be on the border of the minority decision region in the feature-space and only performs SMOTE to oversample those instances, rather than oversampling them all or a random subset.

Cluster-based oversampling [6] (CBOS) attempts to even out the between-class imbalance, as well as the within-class imbalance. There may be subsets of the examples of one class that are isolated in the feature-space from other examples of the same class, creating a within-class imbalance. Small subsets of isolated examples like this are called *small disjuncts*. Small disjuncts often cause degraded classifier performance, and this technique aims to eliminate them without removing data.

RUS was performed at 5%, 10%, 25%, 50%, 75%, and 90% of the majority class. ROS, SM, and BSM were performed with oversampling rates 50%, 100%, 200%, 300%, 500%, 750%, and 1000%. When performing Wilson's editing, we utilized both the weighted and unweighted (standard Euclidean) versions. A total of 31 combinations of sampling technique plus parameter were utilized. In addition, we built a classifier with no sampling, which we denote 'None'. Our research group put forth significant effort to implement the data sampling techniques in software tools to facilitate the experimentation conducted in this work.

4 Experimental Design

4.1 Dataset Description

The CCCS dataset is a military command, control and communications system [7] (publically available at *http://mdp.ivv.nasa.gov*). CCCS (also denoted $\mathcal{C}^o$) has 282 instances, where each instance is an Ada package consisting of one or more methods. Eight software metrics, which are used as independent variables, were collected for CCCS. An additional attribute $nfaults$ (the dependent variable) indicates the number of faults attributed to a program module. Table 1 lists the software metrics in CCCS. Note that it was not the objective of this study to analyze the validity of the software metrics themselves; a different set of metrics can be used in future work. The natural distribution of $nfaults$ in $\mathcal{C}^o$ has over 50% of the program modules with no faults (i.e., $nfaults = 0$), and approximately 19% of the program modules had one fault. The median value of $nfaults$ is 0, the largest value is 42 and the mean is 2.369.

Independent Variables
Logical Operators
Total Lines of Code
Executable Lines of Code
Unique Operands
Total Operands
Unique Operators
Total Operators
Cyclomatic Complexity
Dependent Variable
$nfaults$

Table 1. CCCS Dataset Software Metrics

4.2 Cleansing CCCS

Since $\mathcal{C}^o$ is a real-world dataset, it has some instances which have naturally occurring noisy values for $nfaults$, so-called *inherent* noise. *Injected* noise, by contrast, is noise that is artificially introduced into the dataset. Generating realistic examples of noise in a domain-sensitive manner is a difficult research issue, and is critically important because measuring the results of any technique using unrealistic noise can be misleading since it may not represent the types of noise found in real-world datasets. Therefore, the datasets in our study used a hybrid procedure introduced for cleansing noise from a continuous dependent variable [12], in combination with a software engineering expert, to cleanse a real-world dataset ($\mathcal{C}^o$). In addition to detecting noisy instances, the hybrid procedure also determined a clean value for $nfaults$ (denoted $nfaults^c$) for the instances deemed to be noisy. 81 instances were identified as having inherent noise in $nfaults$ (denoted $nfaults^n$). A detailed description of the cleansing process is presented in our previous work [12], and we cannot include additional details due to space limitations.

4.3 Injected Noise

Two additional datasets were derived from $\mathcal{C}^o$, denoted $\mathcal{C}^{5p}$ and $\mathcal{C}^{10p}$. Note that all three of these datasets have the same number of instances (282) and the same number of attributes (9). These datasets were created as follows:

$\mathcal{C}^{5p}$: The software engineering expert inspected $\mathcal{C}^o$ and identified 14 instances (5% of the dataset) that were relatively clean. These instances were corrupted (with respect to $nfaults$) such that the new $nfaults$ value was noisy but reasonable for the given dataset.

$\mathcal{C}^{10p}$: Starting with $\mathcal{C}^{5p}$, the expert identified and corrupted an additional 14 relatively clean instances in a similar manner, resulting in a total of 28 instances (10% of the dataset) with injected noise in $nfaults$.

$\mathcal{C}^o$, $\mathcal{C}^{5p}$, and $\mathcal{C}^{10p}$ each have 81 inherently noisy examples with respect to $nfaults$. For these 81 instances both the noisy value $nfaults^n$ and clean value $nfaults^c$ are

Dataset	$\#fp$	$\#nfp$	$\%fp$	$\%nfp$
$\mathcal{C}^*_4$	56	226	19.86	80.14
$\mathcal{C}^*_6$	35	247	12.41	87.59
$\mathcal{C}^*_8$	27	255	9.57	90.43
$\mathcal{C}^*_{12}$	19	263	6.74	93.26

Table 2. CCCS Levels of Imbalance in L^c

known (recall the cleansed value was determined by our hybrid procedure in related work [12]). For the injected noise in datasets $\mathcal{C}^{5p}$ and $\mathcal{C}^{10p}$, $nfaults^c$ is the value that was originally in the dataset, and $nfaults^n$ is the corrupted value.

4.4 Level of Imbalance

From $\mathcal{C}^o$, $\mathcal{C}^{5p}$, and $\mathcal{C}^{10p}$, a total of twelve new datasets with a binary class L are derived. Let $\mathcal{C}^*$ denote any of the three initial datasets, and let $\lambda \in \{4, 6, 8, 12\}$ denote a threshold on the dependent variable $nfaults$ in $\mathcal{C}^*$. $\mathcal{C}^*$ is transformed to the dataset $\mathcal{C}^*_\lambda$ by replacing $nfaults$ with a binary class attribute, L. $L(x)$ identifies an instance x as either *fault-prone (fp)* or *not fault-prone (nfp)* according to the following rule:

$$L^\tau_\lambda(x) = \begin{cases} nfp & \text{If } nfaults^\tau(x) < \lambda \\ fp & \text{otherwise} \end{cases}$$

where $\tau \in \{c, n\}$ with $(c = clean, n = noise)$ and $L^\tau_\lambda(x)$ is the class label of x using the clean value $nfaults^c(x)$ or the noisy value $nfaults^n(x)$.

Since an instance x is labeled fp only when $nfaults^\tau(x) \geq \lambda$, increasing the value of λ reduces the number of instances labeled as fp in the dataset, which increases the level of imbalance in L. Four values of λ are used in our experiments, resulting in four levels of imbalance in 12 derived datasets. $\mathcal{C}^o_4, \mathcal{C}^o_6, \mathcal{C}^o_8$, and $\mathcal{C}^o_{12}$ are the four datasets derived from $\mathcal{C}^o$ using the thresholds $\lambda \in \{4, 6, 8, 12\}$. $\mathcal{C}^{5p}_4, \mathcal{C}^{5p}_6, \mathcal{C}^{5p}_8$, and $\mathcal{C}^{5p}_{12}$ were derived from dataset $\mathcal{C}^{5p}$, while $\mathcal{C}^{10p}_4, \mathcal{C}^{10p}_6, \mathcal{C}^{10p}_8$, and $\mathcal{C}^{10p}_{12}$ were derived from dataset $\mathcal{C}^{10p}$. Table 2 shows the number of fp and nfp instances in the data with respect to L^c, the clean class value. L^n is used to construct the models in our experiments, while L^c is used to evaluate the results. In other words, class noise exists only in the training dataset, and not in the test dataset.

4.5 Class Noise

The number of noisy instances in the datasets prior to the transformation of $nfaults$ to L is not necessarily the same as the number of noisy instances in the post-transformation datasets. A class value L will only be incorrect, or noisy, if the value of λ falls between $nfaults^n$ and $nfaults^c$. For example, an instance with $nfaults^n = 7$ and $nfaults^c = 10$ will not be identified as noisy for $\lambda = 6$. Since both $nfaults^n$ and $nfaults^c$ are greater than 6, the instance will be correctly labeled as fp even though $nfaults^n \neq nfaults^c$. However, if $\lambda = 8$ this same instance will be incorrectly labeled as nfp (since $nfaults^n < \lambda$) when it should be labeled as fp (since $nfaults^c \geq \lambda$). Lastly, this instance would be correctly labeled as nfp for $\lambda = 12$, since both $nfaults^n$ and $nfaults^c$ are less than 12.

Dataset	$noise$	$n{\rightarrow}p/p$	$n{\rightarrow}p/n$	$p{\rightarrow}n/n$	$p{\rightarrow}n/p$
$\mathcal{C}^o_4$	10.28	25.45	6.19	6.61	26.79
$\mathcal{C}^o_6$	5.67	22.86	3.24	3.24	22.86
$\mathcal{C}^o_8$	5.67	29.63	3.14	3.14	29.63
$\mathcal{C}^o_{12}$	5.32	37.5	2.28	3.38	47.37
$\mathcal{C}^{5p}_4$	14.54	38.1	10.62	7.76	30.36
$\mathcal{C}^{5p}_6$	9.93	41.86	7.29	4.18	28.57
$\mathcal{C}^{5p}_8$	8.51	45.45	5.88	3.61	33.33
$\mathcal{C}^{5p}_{12}$	6.74	50	3.42	3.79	52.63
$\mathcal{C}^{10p}_4$	19.5	49.32	15.93	9.09	33.93
$\mathcal{C}^{10p}_6$	14.89	56.6	12.15	5.24	34.29
$\mathcal{C}^{10p}_8$	11.7	57.89	8.63	4.51	40.74
$\mathcal{C}^{10p}_{12}$	8.16	59.09	4.94	3.85	52.63

Table 3. Derived Dataset Noise Characteristics (%)

Table 3 shows the percentage of examples with class noise in each of the 12 datasets. Noisy instances are denoted as $x{\rightarrow}y$, where x and y indicate whether L^c and L^n, respectively, belong to the negative (n) or positive (p) class. Column $noise$ shows the percentage of all instances incorrectly labeled. Column $n{\rightarrow}p/p$ shows the percentage of instances labeled fp that should have been labeled nfp. Column $n{\rightarrow}p/n$ shows the instances with $L^c = nfp$ but $L^n = fp$, as a percentage of 'n'. Column $p{\rightarrow}n/n$ shows the percentage of instances labeled nfp that should have been labeled fp. Finally, Column $p{\rightarrow}n/p$ shows the percentage of instances with $L^c = fp$, but $L^n = nfp$. In other words:

$$p{\rightarrow}n/p = \frac{\#p{\rightarrow}n}{\#p{\rightarrow}p + \#p{\rightarrow}n} \quad (1)$$

$$p{\rightarrow}n/n = \frac{\#p{\rightarrow}n}{\#n{\rightarrow}n + \#p{\rightarrow}n} \quad (2)$$

$$n{\rightarrow}p/n = \frac{\#n{\rightarrow}p}{\#n{\rightarrow}n + \#n{\rightarrow}p} \quad (3)$$

$$n{\rightarrow}p/p = \frac{\#n{\rightarrow}p}{\#p{\rightarrow}p + \#n{\rightarrow}p} \quad (4)$$

Note that the level of imbalance and amount of class noise are inversely correlated - as the imbalance increases, the level of noise decreases. This effect is caused by the noise simulation strategy used in this work. As λ increases, there are fewer examples with $nfaults^c < \lambda$ but $nfaults^n > \lambda$ (or $nfaults^c > \lambda$ but $nfaults^n < \lambda$). Our experiments utilized this methodology for the creation of datasets because classification problems in software engineering often require the conversion of the number of faults to a binary class using a threshold [8]. It was our objective to analyze the impacts of varying the threshold used for de-

sampling	$\lambda = 4$	$\lambda = 6$	$\lambda = 8$	$\lambda = 12$	Average
BSM	0.9469	0.9523	0.9637	0.934	0.9492
CBOS	0.8932	0.9153	0.9145	0.8904	0.9033
OSS	0.9421	0.9309	0.9371	0.9142	0.931
ROS	0.9422	0.9516	0.9513	0.9248	0.9425
RUS	0.9476	0.9598	0.9632	0.9598	0.9576
SM	0.9476	0.9549	0.9586	0.9352	0.9491
WE	0.949	0.9582	0.9597	0.9367	0.9509
None	0.9429	0.9392	0.9421	0.9229	0.9368
Avg	0.9389	0.9453	0.9488	0.9273	–

Table 4. Average AUC values for $noise = O$

sampling	$\lambda = 4$	$\lambda = 6$	$\lambda = 8$	$\lambda = 12$	Average
BSM	0.924	0.9229	0.9378	0.925	0.9274
CBOS	0.8276	0.8411	0.8804	0.8594	0.8522
OSS	0.9278	0.912	0.9258	0.9134	0.9198
ROS	0.9072	0.9154	0.9359	0.9159	0.9186
RUS	0.932	0.9344	0.9537	0.9436	0.9409
SM	0.9191	0.9205	0.9346	0.9157	0.9225
WE	0.9276	0.9339	0.95	0.9277	0.9348
None	0.9241	0.9148	0.9303	0.9164	0.9214
Avg	0.9112	0.9119	0.9311	0.9147	–

Table 5. Average AUC values for $noise = 5P$

sampling	$\lambda = 4$	$\lambda = 6$	$\lambda = 8$	$\lambda = 12$	Average
BSM	0.8915	0.8807	0.9127	0.9089	0.8985
CBOS	0.7674	0.7872	0.8476	0.8334	0.8089
OSS	0.9014	0.881	0.9079	0.8991	0.8973
ROS	0.8764	0.8762	0.9123	0.8995	0.8911
RUS	0.9076	0.9053	0.936	0.9321	0.9203
SM	0.8888	0.8817	0.9134	0.8993	0.8958
WE	0.9024	0.8934	0.9308	0.9242	0.9127
None	0.8987	0.8771	0.9051	0.9022	0.8958
Avg	0.8793	0.8728	0.9082	0.8998	–

Table 6. Average AUC values for $noise = 10P$

termining the class when confronted with realistic examples with class noise (which is why both our noise cleansing and injection strategies were overseen by a software engineering expert).

4.6 Classifier Evaluation

We utilize a widely-known performance metric called the area under the ROC curve (AUC) to evaluate learner performance in our experiments. The ROC curve graphs the true positive rate on the y-axis versus the false positive rate on the x-axis, and therefore measures the tradeoff between detection rate and false alarm rate. The AUC ranges from zero to one, with higher values denoting a classifier with generally better performance (in general, a higher AUC implies a higher true positive rate with a lower false positive rate, which is preferred in most applications). Two different classifiers can be evaluated by comparing their AUC values. Provost and Fawcett [15] give an extensive overview of ROC curves and their potential use for optimal classification.

4.7 Summary of the Experimental Design

Learner performance is measured using 10-fold cross validation (CV) with 30 independent replications for each experiment, making all experiments performed in this work comprehensive and statistically significant. With 10-fold CV, nine folds are used as a training dataset and one fold is the hold-out (test) dataset. When applying sampling techniques and constructing classifiers using the training dataset, the noisy label L^n is used, as the objective of our study is to examine the impact of class noise on learning from imbalanced data using data sampling. While the training dataset contains class noise, the test dataset does not. Therefore, each classifier's predictions are compared to the clean value, L^c, when calculating the AUC. A total of 1,267,200 learners (software quality classification models) were constructed in these experiments.

5 Empirical Results

In this section, we examine the effects of $noise$ and $imbalance$ on the performance of the seven sampling techniques: BSM, CBOS, OSS, ROS, RUS, SM, and WE. In addition, we compare the performance of each sampling technique to performance achieved by not performing any data sampling (denoted as None). We begin by examining the effect of noise on sampling technique performance in Section 5.1, and then take a closer look at the combined effect of noise and imbalance in Section 5.2. The data used for this analysis is in Tables 4 through 6. These tables show the average AUC across all 11 learners for each sampling technique on each dataset. Individual learner results are not provided due to space restrictions. In cases where multiple parameters were used for sampling, the results are based on the parameter that achieved the best performance. This was done because it was not the objective of this work to analyze the impact of varying parameters on the performance of sampling techniques (we leave this to future work and cannot discuss here due to space limitations). Finally, a practitioner can evaluate and select an appropriate value for the parameter settings using cross validation when building software quality classification models in practice, and hence it is reasonable to report the results based on a value that gives good performance.

5.1 Sampling Technique and Noise Level

Tables 4 through 6 show the results grouped by $noise$ level. The noise levels $O, 5P$, and $10P$ indicate that the results were obtained from datasets derived from $\mathcal{C}^o$, $\mathcal{C}^{5p}$, and $\mathcal{C}^{10p}$, respectively. Recall that O is the least noisy dataset, followed by $5P$ and $10P$. The results in Table 4 are for $noise = O$. Tables 5 and 6 provide results for $noise = 5P$ and $10P$, respectively. The first column of each of these

tables lists the $sampling$ technique. Columns two through five list the results for each level of imbalance. The last column shows the average AUC for each sampling technique across all imbalance levels for the specified level of noise. For O in Table 4, RUS achieves the highest average AUC (0.9576), followed by WE (0.9509). For noise levels $5P$ and $10P$, RUS and WE remain the two best techniques, followed by BSM. CBOS performs poorly for all three levels of noise. Averaged over all of the levels of imbalance, the AUC for all sampling techniques deteriorates with higher levels of noise (comparing the last column of Tables 4 through 6).

5.2 Combined Effect of Noise and Imbalance on Sampling Techniques

Section 5.1 examined the effect of noise on sampling techniques independent of the level of imbalance in the data. This section considers the combined effect of the level of noise and the level of imbalance in a dataset on these techniques.

Figures 1 and 2 show the performance (AUC) of each sampling technique as noise is increased at imbalance levels $\lambda = 4$ and $\lambda = 12$, respectively. The results of None (not performing sampling) are included in each plot to serve as a base-line for comparison, as well as to show whether the sampling technique provides improved performance over not performing any sampling. CBOS, which was outperformed by every other sampling technique, is not included in these figures.

Figure 1 shows the results for $\lambda = 4$. At this level of imbalance, BSM does not outperform None at the lowest level of noise. However, as the level of noise is increased, BSM performs slightly better than None. OSS results in a lower AUC than None at the lowest level of noise, but at $noise = \{5P, 10P\}$ OSS results in a higher AUC. ROS and None show similar performance at the lowest level of noise, but as noise is increased, the performance of ROS is much worse than None, suggesting that ROS is severely affected by the presence of class noise. SM achieves a higher AUC than None at the lowest level of noise, but as noise is increased its performance drops more rapidly than None, resulting in lower AUC values for $noise = \{5P, 10P\}$. RUS and WE both show consistent improvement over None at all levels of noise. In general, given a threshold of $\lambda = 4$ and at different levels of class noise, sampling did not dramatically improve learner performance (and in some cases such as ROS actually harmed performance).

Figure 2 shows the results for $\lambda = 12$. Unlike imbalance level $\lambda = 4$, at this level of imbalance BSM outperforms None at all levels of noise, suggesting that BSM does improve classification performance in highly imbalanced and noisy datasets. On the other hand OSS, which improved performance at higher noise levels for $\lambda = 4$, results in a lower AUC for all levels of noise, suggesting that OSS does not perform as well if the data is both noisy and highly imbalanced. ROS, which was severely outperformed by None at $\lambda = 4$, achieves an AUC that is similar to no sampling at all noise levels. SM, which outperformed None at high levels of noise for $\lambda = 4$, only achieves a higher AUC for $noise = O$ at this level of imbalance ($\lambda = 12$). RUS and WE again outperform None at all levels of noise. At this level of imbalance, RUS outperforms None by a much larger margin than with $\lambda = 4$. WE also shows a larger improvement in performance compared to None when the imbalance level increased from $\lambda = 4$ to $\lambda = 12$. Also worth noting is that the line representing the performance of WE is relatively horizontal, suggesting that this technique is highly robust to the presence of noise, especially when data is severely imbalanced. This is not surprising since WE is designed to avoid including noisy examples in the post-sampling training data.

5.3 Threats to Validity

Experimental research in empirical software engineering commonly includes a discussion of two different types of threats to validity [20]. Threats to internal validity relate to unaccounted influences that may impact the empirical results. Threats to external validity consider the generalization of the results outside the experimental setting, and what limits, if any, should be applied.

From the perspective of internal validity, great care was taken to ensure the legitimacy of our experimental results. The benchmark WEKA data mining tool [19] was used for the construction of all classifiers, and we have included all of the parameter settings used to ensure repeatability. The parameters for the learners were chosen to ensure good performance in many different circumstances and to be reasonable for the datasets.

In relation to external validity, we believe that the results presented in this work generalize to other software measurement datasets. Future work will include similar studies using additional datasets to confirm this belief. CCCS is a real-world software measurement dataset, and noise simulation, overseen by a software engineering expert with over 15 years of domain experience, was performed to ensure that it was reasonable for the dataset. One of the strengths of our work is to use datasets derived from real-world data, which as mentioned is in contrast to many of the previous studies on data quality, where simulated data is often used.

6 Conclusion

We have presented an experimental study of the effects of imbalance and noise on different sampling techniques in software measurement data. Using 12 measurement datasets with different levels of noise and imbalance, we compared the performance of seven sampling techniques

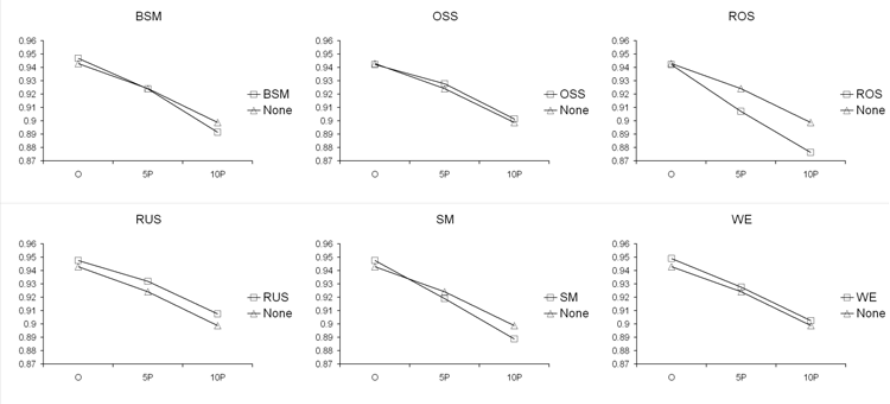

Figure 1. Sampling technique performance as noise increases: $\lambda = 4$

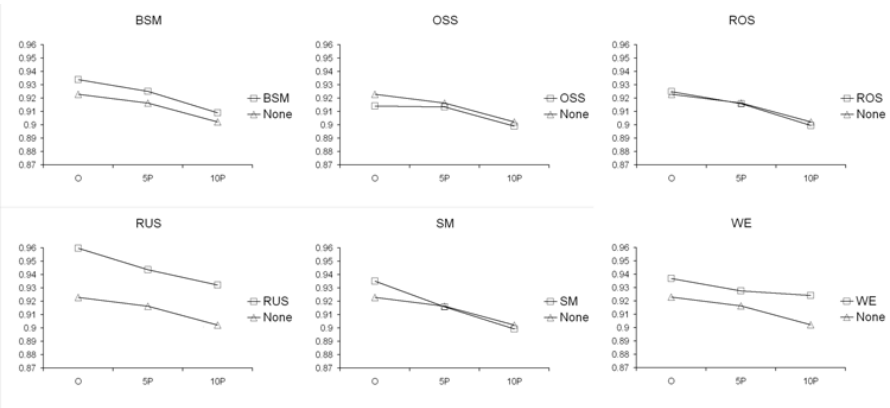

Figure 2. Sampling technique performance as noise increases: $\lambda = 12$

(plus no sampling) using 11 classification algorithms on imbalanced and noisy data. Based on our empirical evaluation we draw the following conclusions:

1. How are data sampling techniques affected by the presence of noise in software quality data? What benefit do these techniques provide when applied to a noisy dataset?

   (a) Some sampling techniques are relatively more affected by noise than others.

   (b) RUS, WE, and BSM are each relatively robust in the presence of noise. At all levels of noise, these three sampling techniques outperformed the remaining four techniques and provided better results than when sampling is not performed (None).

   (c) ROS and SM perform well at low levels of noise, but are not very robust. Their performance is worse than None when the level of noise is high.

   (d) Although the overall performance of OSS is not as good as many of the other sampling techniques, it is relatively robust to noise.

   (e) CBOS did not perform well in any of our experiments.

2. How does the combination of imbalance level and noise level affect these data sampling techniques? Are sampling techniques more affected by noise when imbalance is more severe?

   (a) The level of imbalance in the data affects sampling technique performance as the level of noise is increased.

   (b) BSM, which does not outperform None at low levels of imbalance, shows significant improvement over None at all noise levels when the data is more severely imbalanced.

   (c) OSS which (slightly) outperforms None at low levels of imbalance and high levels of noise does not perform as well when the level of imbalance is high.

   (d) RUS and WE both outperform None by a larger margin at high levels of imbalance and all levels of noise. WE is relatively robust to noise, especially at higher levels of imbalance.

Future work will extend this study by using additional software measurement datasets to verify our conclusions. In addition, this work will be extended by applying noise handling techniques in conjunction with sampling techniques to

enhance software quality classification performance. In doing so, we will study the impact of class imbalance on these noise handling techniques. Finally, future work will also analyze the performance of each learner separately, which could not be included due to space limitations.

References

[1] R. Barandela, R. M. Valdovinos, J. S. Sanchez, and F. J. Ferri. The imbalanced training sample problem: Under or over sampling? *In Joint IAPR International Workshops on Structural, Syntactic, and Statistical Pattern Recognition (SSPR/SPR'04), Lecture Notes in Computer Science 3138*, (806-814), 2004.

[2] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[3] 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.

[4] 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*, 2003.

[5] 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.

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

[7] T. M. Khoshgoftaar and E. B. Allen. Classification of fault-prone software modules: Prior probabilities, costs and model evaluation. *Empirical Software Engineering*, 3:275–298, 1998.

[8] 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.

[9] T. M. Khoshgoftaar and N. Seliya. Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Software Engineering Journal*, 9(2):229–257, 2004.

[10] T. M. Khoshgoftaar and N. Seliya. The necessity of assuring quality in software measurement data. In *Proceedings of 10th International Software Metrics Symposium*, pages 119–130, Chicago, IL, September 2004. IEEE Computer Society.

[11] T. M. Khoshgoftaar and J. Van Hulse. Empirical case studies in attribute noise detection. In *Proceedings of the IEEE International Conference Information Reuse and Integration*, pages 211–216, Las Vegas, NV, August 2005.

[12] T. M. Khoshgoftaar, J. Van Hulse, and C. Seiffert. A hybrid approach to cleansing software measurement data. In *Proceedings of the* 18^{th} *IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2006)*, pages 713–722, Washington, D.C., November 13-15 2006.

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

[14] M. Maloof. Learning when data sets are imbalanced and when costs are unequal and unknown. In *Proceedings of the ICML'03 Workshop on Learning from Imbalanced Data Sets*, 2003.

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

[16] J. Van Hulse, T. M. Khoshgoftaar, and H. Huang. The pairwise attribute noise detection algorithm. *Knowledge and Information Systems Journal, Special Issue on Mining Low Quality Data*, 11(2):171–190, 2007.

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

[18] G. M. Weiss and F. Provost. Learning when training data are costly: the effect of class distribution on tree induction. *Journal of Artificial Intelligence Research*, (19):315–354, 2003.

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

[20] 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.

[21] X. Zhu and X. Wu. Class noise vs attribute noise: A quantitative study of their impacts. *Artificial Intelligence Review*, 22(3-4):177–210, November 2004.