Information Sciences 181 (2011) 4867–4877

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Class noise detection based on software metrics and ROC curves

Cagatay Catal [*], Oral Alan, Kerime Balkan

The Scientific and Technological Research Council of Turkey (TUBITAK), Center of Research for Advanced Technologies of Informatics and Information Security, Information Technologies Institute, 41470 Gebze, Kocaeli, Turkey

A R T I C L E I N F O

Article history:
Received 17 October 2009
Received in revised form 2 June 2011
Accepted 15 June 2011
Available online 2 July 2011

Keywords:
Receiver Operating Characteristic (ROC) curve
Noise detection
Software metrics
Metric threshold values
Software fault prediction
Software quality

A B S T R A C T

Noise detection for software measurement datasets is a topic of growing interest. The presence of class and attribute noise in software measurement datasets degrades the performance of machine learning-based classifiers, and the identification of these noisy modules improves the overall performance. In this study, we propose a noise detection algorithm based on software metrics threshold values. The threshold values are obtained from the Receiver Operating Characteristic (ROC) analysis. This paper focuses on case studies of five public NASA datasets and details the construction of Naive Bayes-based software fault prediction models both before and after applying the proposed noise detection algorithm. Experimental results show that this noise detection approach is very effective for detecting the class noise and that the performance of fault predictors using a Naive Bayes algorithm with a logNum filter improves if the class labels of identified noisy modules are corrected.

1. Introduction

Examples of data quality problems in real-world datasets include incomplete, noisy, and inconsistent data [7], and this low-quality data lowers the performance of machine learning-based models. Preprocessing in data mining, which is an important step in the data mining process, improves the data quality. Data cleaning, data integration, data transformation, and data reduction routines are the major tasks of data preprocessing. These terms are defined as follows:

- *Data cleaning* (*or data cleansing*): Missing values are filled in; outliers are identified; noisy data is removed; and inconsistent data is corrected.
- *Data integration:* Data from multiple databases or data cubes is combined into a data store.
- *Data transformation:* Data is transformed into appropriate forms using normalization or aggregation techniques.
- *Data reduction:* Data is reduced to a smaller volume, but the new dataset provides similar analytical results. Attribute subset selection, dimensionality reduction, numerosity reduction, and data cube aggregation are some strategies used for data reduction [7].

The following five types of data can be described for a dataset: mislabeled cases, redundant data, outliers, borderlines, and safe cases [16]. Mislabeled cases are noisy data objects that introduce *class noise*. Redundant data form clusters and those entries can be removed from the dataset since some of the clusters can be represented by other clusters. Outliers are significantly dissimilar to the other data objects in the dataset, and they may be noisy instances or very extreme cases. Borderline

[*] Corresponding author.
E-mail address: cagatay.catal@bte.tubitak.gov.tr (C. Catal).

0020-0255/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2011.06.017

instances are very near to decision borders, and they are not considered as reliable as safe cases. Safe cases represent the remaining data points that should be used in the learning phase [16].

Although many researchers use the keywords "outlier" and "noisy instances" interchangeably, outlier and noisy instances actually refer to two different concepts that are used in data mining. While outliers are interesting to the analyst, noise that is present in data does not provide any significance by itself, yet the interest in outliers exempts it from noise removal [3]. For credit card fraud detection, these kinds of instances are very valuable because such instances may point to malicious activities [3]. Therefore, these instances are outliers for the credit card fraud detection domain. For the software engineering domain, these kinds of instances do not provide any interesting information because they are mostly caused by human error during the labeling process of modules. Therefore, these approaches to software engineering should be evaluated from the perspective of noise detection rather than outlier detection. Numerous algorithms have been developed to cleanse software measurement datasets within the last decade [29,6,24].

Data noise can be categorized into the following two groups: class and attribute noise. "Class noise occurs when an error exists in the class label of an instance, while attribute noise occurs when the values of one or more attributes of an instance are corrupted or incorrect" [11]. Most data mining studies have focused on detecting class noise, and more recent studies have shown that class noise impacts classifiers more severely than attribute noise [33]. Developers may not report each software fault that occurs during software tests, since a larger number of faults may create the perception of poor performance on the part of the developers [29]. In addition, developers may consider some faults as not significant for the project; thus, they may not report this kind of fault. As such, some software modules that should be labeled as faulty are labeled as non-faulty during the labeling process due to the inaccurate reporting of faults. In addition, data entry and data collection errors may also cause class and attribute noise [10].

Many empirical studies have concluded that the presence of class noise in the dataset adversely impacts the performance of machine learning-based classifiers [33,1,6,29]. In addition, the presence of attribute noise decreases the performance of classifiers as well [33]. Therefore, the preprocessing task of identifying noisy instances prior to the training phase of classifiers is a significant issue that should be addressed in a data mining initiative. Another solution used to prevent the degradation of classifier performance is the development of classifiers that are robust and not sensitive to noise. Robustness to noise is directly related to the machine learning algorithm. Van Hulse and Khoshgoftaar [29] reported that simple algorithms, such as the Naive Bayes algorithm, were more resistant to the impact of class noise than complex classifiers, such as the Random Forests algorithm. In addition, they showed that the noise resulting from the minority class is more severe than the noise resulting from the majority class. Therefore, faulty modules that are incorrectly labeled as non-faulty modules cause the most severe type of noise in software measurement datasets.

In this study, we developed a noise detection approach that uses threshold values for software metrics in order to capture these noisy instances. These thresholds were calculated using Shatnawi et al.'s [26] threshold calculation technique. We empirically validated the proposed noise detection technique on five public NASA datasets and built Naive Bayes-based software fault prediction models both before and after applying the proposed noise detection algorithm. After class labels of noisy instances were corrected, the performance of the fault predictors increased significantly. This observation and other results of experiments show that this technique is very effective for detecting class noise in software measurement datasets.

Twenty-one metrics are shown in NASA datasets, but we used thirteen software metrics for experiments since the remaining metrics are derived from four Halstead primitive metrics. Folleco et al. [6] used the same thirteen metrics to investigate the impact of increasing levels of class noise on a software quality classification problem. Khoshgoftaar and Van Hulse [11] applied the same metrics set to validate their attribute noise detection technique with NASA datasets.The threshold calculation approach used in this study is slightly different from Shatnawi et al.'s technique. They stated that they chose the candidate threshold value that had maximum values of both sensitivity and specificity. However, such a candidate threshold may not always exist. Therefore, we calculated the area under the ROC curve (AUC) that passes through three points (0,0), (1,1), and (PD,PF) where PD is the probability of detection and PF is the probability of false alarm. We chose the candidate threshold value that provides the maximum AUC. Details of the proposed approach are given in Section 3.1.

A two-stage algorithm is used for noise detection approach. Noisy instances were identified with noise detection approach and their labels were corrected before building the fault prediction model.The motivation behind experiments comes from the recently published related studies that are presented in Section 2. Recently, a number of techniques were developed to identify attribute and class noise. However, none of these noise detection techniques considered the use of threshold values for software metrics, even though software metrics are widely used in software quality prediction studies. Software quality prediction is a challenging research area in the field of software engineering [22]. In this study, we showed that the threshold values for software metrics can be identified using ROC analysis and that the performance of fault predictors using the Naive Bayes algorithm and logNum filter significantly improves if the class labels of identified noisy modules are corrected. The remainder of this paper is structured as follows:

- Section 2 presents related work;
- Section 3 introduces the proposed noise detection approach;
- Section 4 presents empirical case studies using real-world data from NASA projects; and
- Section 5 contains the conclusion and suggestions for future work.

2. Related work

Recently, several studies investigated the impact of class and attribute noise on software quality classification problems, and researchers have proposed several approaches to cleansing software measurement datasets. Learning from both imbalanced and noisy data is challenging, and some notable papers in software engineering literature discuss this type of learning.

As previously mentioned, Van Hulse and Khoshgoftaar [29] showed that class noise degrades the performance of machine learning algorithms on imbalanced datasets and that the most severely distracting type of datanoise is that resulting from the minority class. Also, they concluded that simple algorithms, such as the Naive Bayes, are more resistant to noise than more complex machine learning algorithms, such as Random Forests. Folleco et al. [6] reported that increasing levels of class noise adversely impact classification performance and that noise coming from the minority class is more damaging than noise coming from the majority class. When the percentage of noise from the majority class was 50%, the classifiers consistently achieved very good performance results since no noise came from the minority class [6]. However, the impact of minority class noise was detrimental in their experiments. The results of Naive Bayes, C4.5, and Random Forest algorithms,used on seven datasets injected with class noise, were analyzed. They concluded that the best and most consistent classification performance is achieved when using the Random Forest algorithm. This result contradicts that of Van Hulse and Khoshgoftaar [29], which indicated that Naive Bayes is often more robust than Random Forest.

This research focused on the data noise problem in software measurement datasets; the impact of missing values was not investigated in noisy datasets. Van Hulse and Khoshgoftaar [28] empirically evaluated the missing value imputation in noisy software measurement data. They concluded that Bayesian multiple imputation and regression imputation are very effective for addressing the missing attribute problem, and that noisy instances adversely impact the imputation process. One solution to address missing values is to remove data from the dataset, but this method can remove valuable information. Zhong et al. [32] reported that the JM1 dataset, i.e., a public dataset from NASA in the PROMISE repository, includes inconsistent modules (i.e., the same attribute values but different fault labels) and missing values. They removed 2035 modules due to inconsistencies prior to the execution of algorithms, but this possibly caused the loss of valuable information for the analysis. Therefore, a missing value imputation in a noisy dataset has significant consequences, and new methods, tools or techniques must be developed in order to prevent the removal of modules that have missing values.

Khoshgoftaar and Van Hulse [11] developed a methodology for attribute noise detection, and this technique—using the Pairwise Attribute Noise Detection algorithm (PANDA)—generates a ranking of attributes from most to least noisy. Khoshgoftaar et al. [10] proposed a noise detection approach based on Boolean rules generated from the software measurement dataset and concluded that this approach is simpler and more attractive than noise filtering techniques such as the Ensemble Filter. The performance of a C4.5-based classification filter was worse than this approach in detecting noisy instances, and the performance of this approach improved when the number of noisy attributes was increased.

Khoshgoftaar et al. [13] developed a hybrid technique to cleanse the dataset, and the cleaning was limited to the dependent variable that showed the number of faults. The performance of multivariate linear regression models on a cleansed dataset was better as compared to the performance of such models on the original dataset. This observation confirmed that the new dataset is cleaner than the original one. In this study, the dependent variable was the number of faults instead of the Boolean fault label (i.e., faulty or non-faulty). Seiffert et al. [24] investigated the classification performance of learners on imbalanced and noisy data when seven sampling techniques were used for the dataset, and they concluded that some sampling techniques are more robust to noise. Therefore, sampling techniques should be considered in light of noisy instances when attempting to solve software quality classification problems.

Khoshgoftaar and Seliya [9] suggested focusing on improving the quality of datasets instead of looking for the best classifier on software measurement datasets. They reported that even the best classification technique can provide very poor performance results if the quality of the dataset is very low. Yoon et al. [30] applied a K-means clustering method to detect outliers and reported that 11.5% of the dataset included outliers. A Cubic Clustering Criterion (CCC) was used to calculate the K value using the SAS Enterprise Miner 4.3 commercial tool, and a dataset collected within two years is from a financial company in Korea. Both external and internal outliers were defined in this study. Khoshgoftaar et al. [8] showed that rule-based modeling (RBM) is very effective in determining the outliers and that the performance of a case-based reasoning (CBR) model degrades when outliers are used in the dataset. Therefore, CBR is clearly sensitive to outliers. Khoshgoftaar et al. [12] developed an ensemble filter that includes twenty-five classification techniques, and instances that were incorrectly classified by the majority of classifiers are labeled as noisy. Li et al. [15] developed a fuzzy relevance vector machine for learning from imbalanced and noisy data. Zhu and Wu [33] showed that class noise impacts classifiers more severely than attribute noise.

3. Noise detection approach

This section presents methods for calculating the threshold values of software metrics by using ROC curves and introduces the proposed noise detection approach, which uses these threshold values.

3.1. Method to find the threshold values of software metrics

In order to identify risky software modules, the threshold values of software metrics are easier to use in practice as compared to building a model based on metrics [26]. However, finding the threshold values of the metrics is not an easy task, and few researchers have proposed threshold calculation techniques to date. Ulm's logistic regression method [27] was used by El Emam et al. [4], but they reported that the threshold values calculated using Ulm's method were not valid. Erni and Lewerentz [5] used the average and standard deviations of metrics to calculate threshold values, but they did not consider the fault-proneness labels of data points. Therefore, it is not clear whether these values are associated with error-proneness. Rosenberg [23] presented some threshold values calculated by means of a histogram analysis for object-oriented Chidamber–Kemerer (CK) metrics in order to find classes for inspection, but we do not know how these values are associated with error probability [26]. A recent technique to find threshold values was proposed by Shatnawi et al. [26], and this technique uses ROC curves that are widely used in data mining for performance evaluation. On an ROC curve, the probability of a false alarm (PF) is plotted on the x-axis, and the probability of detection (PD) is plotted on the y-axis. However, the threshold calculation approach differs slightly from their technique. They stated that they chose the candidate threshold value that has the maximum value for both sensitivity and specificity, but such a candidate threshold may not always exist. We calculated the AUC of the ROC curve that passes through three points, i.e., (0,0), (1,1), and (PD,PF), and we chose the threshold value that maximizes the AUC. A ROC curve is shown in Fig. 1. ROC curves and AUC are widely used in machine learning for performance evaluation [18,2,17].

ROC curves pass through the points (0,0) and (1,1) by definition, and the third point (PD,PF) is calculated for each candidate threshold value. The interval for the candidate threshold values is between the minimum and maximum value of that metric in the dataset. A confusion matrix, and thus the (PD,PF) point, is calculated for each candidate threshold value. A confusion matrix for each candidate threshold value is calculated by using Table 1. Each data point in the dataset is predicted as non-faulty if the data point's metric value is lower than the candidate threshold value. Otherwise, the data point is predicted as faulty, as shown in Table 1. Since we know the actual labels of data points in the datasets, we calculate the (PD,PF) pair for each candidate threshold value.

Equations that were used to calculate PD and PF values are shown in Eqs. 1 and 2, respectively. "Each pair, (PD,PF), represents the classification performance of the threshold value that produces this pair. The pair that produces the best classification performance is the best threshold value to use in practice" [26].

$$\mathrm{PD} = \frac{TP}{TP + FN} \tag{1}$$

$$\mathrm{PF} = \frac{FP}{FP + TN} \tag{2}$$

Shatnawi et al. [26] reported that they chose the candidate threshold value with the maximum value for PD and the minimum value for PF as the best threshold value. However, we noticed that such a candidate threshold may not always exist

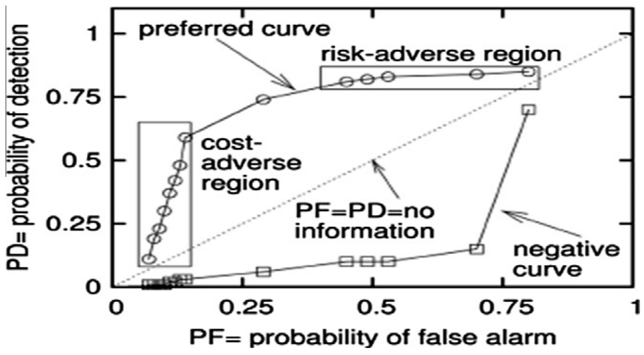


Fig. 1. ROC curve [19].

Table 1
The confusion matrix based on a candidate threshold value.

	Actual labels	
	Faulty	Non-faulty
Predicted labels		
Metric >= threshold	True-positive (TP)	False-positive (FP)
Metric < threshold	False-negative (FN)	True-negative (TN)

because of the trade-off between PD and PF values, and one candidate threshold value may have the highest value for PD, while another candidate threshold value may have the lowest value for PF. In order to fill this gap, as previously mentioned, we used AUC for the ROC curve that passes through the following three points: (0,0), (1,1) and (PD,PF). The candidate threshold value that maximizes the AUC is the best threshold value, according to the threshold calculation technique. Furthermore, the AUC value of the best threshold value should normally be higher than 0.5. We calculated each metric threshold value by using this technique.

After we identified the threshold values for the metrics, we used these values in the noise detection approach, which is explained in the next section.

3.2. Noise detection technique

The proposed noise detection approach involves a two-stage algorithm. First, it detects noisy data objects that have non-faulty class labels. *A data object that has a non-faulty class label is considered a noisy instance if the majority of the software metric values exceed their corresponding threshold values.* At the second stage, it detects noisy data objects that have faulty class labels. *A data object that has a faulty class label is considered a noisy instance if all of the metric values are below their corresponding threshold values.*

Before building this noise detection method, we examined the features of each module in NASA datasets that are located in the PROMISE repository. We noticed that several modules had very low metric values but also had faulty class labels. Furthermore, some modules had very high metric values but also had non-faulty class labels. These observations helped us to construct the two-stage noise detection approach that uses both class labels and software metrics.

In this section, we elaborate on how we obtained and evaluated these two rules, as the rules themselves are at the core of the original contribution of this paper. As easily seen from the explanation of this technique, the definition of noisy objects is not symmetrical. At first, we decided to define extreme rules to identify the noisy instances, and the second rule created by means of this technique laid down the following limit: "*A data object that has a faulty class label is considered a noisy instance if* ***all of the metric values*** *are below their corresponding threshold values.*" If the definition of noisy objects were symmetrical, we would define the first rule as follows: "*A data object that has a non-faulty class label is considered a noisy instance if* ***at least one*** *of the software metric values exceeds its corresponding threshold value.*" However, this definition would not combine different sources of information (or metrics) and it would trust only a simple cut-off value of a specific metric. Therefore, we aimed to combine several metrics by using the *majority* term in order to consider a non-faulty instance as noisy; hence, the first rule arose.

Experimental studies showed that this noise detection technique is very effective for detecting noisy modules. After this analysis, we needed to see how the performance of the approach would change if the rules were made symmetrical. When we changed the first rule as explained above in order to make it symmetrical, the performance variation was not deterministic and such a noise detection approach was not successful. For this reason, we did not change the rules and used them as-is, even though they are not symmetrical.

After we identified noisy modules, we corrected their class labels, and the numbers of total class label changes are shown in Table 2. Calculated threshold values are shown in Tables 3 and 4. The LOC, v(g), eV(g), and iv(g) columns represent McCabe's lines of code metric, cyclomatic complexity, essential complexity, and design complexity, respectively. loCode, loComment, loBlank, locCodeandComment, uniq_Op, uniq_Opnd, total_Op, total_Opnd, and Branch count columns represent

Table 2
The numbers of total class label changes.

Datasets	True to false	False to true	Total changes
KC1	31	442	473
KC2	16	73	89
CM1	5	136	141
PC1	15	276	291
JM1	486	1820	2306

Table 3
Software metric threshold values for datasets.

Datasets	LOC	v(g)	eV(g)	iv(g)	lOCode	loComment	lOBlank
KC1	11	3	2	3	5	1	1
KC2	32	5	4	3	24	1	6
JM1	33	5	4	3	27	4	4
CM1	28	3	2	3	2	7	7
PC1	22	4	4	3	22	4	1

Table 4
Remaining software metric threshold values for datasets.

Datasets	locCodeandComment	uniq_Op	uniq_Opnd	total_Op	total_Opnd	Branch count
KC1	1	7	7	13	8	4
KC2	1	12	17	42	29	4
JM1	1	15	19	54	36	8
CM1	0	18	21	53	27	5
PC1	7	15	20	50	34	7

Halstead's line count, count of lines of comments, count of blank lines, count of lines of code and comment, unique operator, unique operand, total operator, total operand, and branch count, respectively.

We only used thirteen software metrics for experiments since the remaining metrics in the datasets are derived from these thirteen primitive metrics.

4. Empirical case studies

This section presents the datasets, performance evaluation metrics, machine learning algorithm, and filtering techniques that were used in this study and the results of the experimental studies.

4.1. System description

NASA projects located in the PROMISE repository were used for experiments. We used five public datasets called JM1, KC1, PC1, KC2, and CM1. JM1 belongs to a real-time predictive ground system project and consists of 10,885 modules. This dataset has noisy measurements, and 19% of the modules are defective. JM1 is the largest dataset in experiments and uses the C programming language. As previously mentioned, Seliya et al. [25] reported that the JM1 dataset contains some inconsistent modules (i.e., identical metrics but different labels). They removed 2035 modules (i.e., 19% of the dataset) due to the inconsistency problem prior to the execution of algorithms, and worked with 8850 modules. However, we are not sure if they are truly noisy because the NASA quality assurance group reported these faults and metrics. For this reason, we did not eliminate such modules as most of the studies in literature have done. As indicated by the JM1 example, these public datasets may contain inconsistent or noisy modules. The KC1 dataset, which uses the C++ programming language, belongs to a storage management project for receiving/processing ground data. It has 2109 software modules, 15% of which are defective. PC1 belongs to a flight software project for an Earth-orbiting satellite. It has 1109 modules, 7% of which are defective, and uses the C implementation language. The KC2 dataset belongs to a data processing project, and it has 523 modules. The C++ language was used for the KC2 implementation, and 21% of its modules have defects. CM1 belongs to a NASA spacecraft instrument project and was developed in the C programming language. CM1 has 498 modules, of which 10% are defective.

4.2. Performance evaluation parameters

In this study, we used the False Positive Rate (FPR), False Negative Rate (FNR), and Error parameters to evaluate the performance of noise detection algorithm. We expected error rates to decrease when the class labels of noisy instances were corrected. If the error rates decreased, this shows that the performance of the fault prediction models increased.Error is the percentage of mislabeled modules. FPR is the percentage of non-faulty modules labeled as faulty by the model, and FNR is the percentage of faulty modules labeled as non-faulty.

Table 5 shows a sample confusion matrix and is explained below:

- The actual class label of the module is faulty, and the prediction result is faulty (TP).
- The actual class label of the module is faulty, but the prediction result is non-faulty (FN).
- The actual class label of the module is non-faulty, but the prediction result is faulty (FP).
- The actual class label of the module is non-faulty, and the prediction result is non-faulty (TN).

Table 5
A sample confusion matrix.

	(Prediction) Non-faulty	(Prediction) Faulty
(Actual) Non-faulty	True negative (TN)	False positive (FP)
(Actual) Faulty	False negative (FN)	True positive (TP)

Eqs. (3)–(5) are used to calculate the evaluation parameters, and they are as follows:

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} \tag{3}$$

$$\mathrm{FNR} = \frac{\mathrm{FN}}{\mathrm{FN} + \mathrm{TP}} \tag{4}$$

$$\mathrm{Error} = \frac{\mathrm{FN} + \mathrm{FP}}{\mathrm{TP} + \mathrm{FP} + \mathrm{FN} + \mathrm{TN}} \tag{5}$$

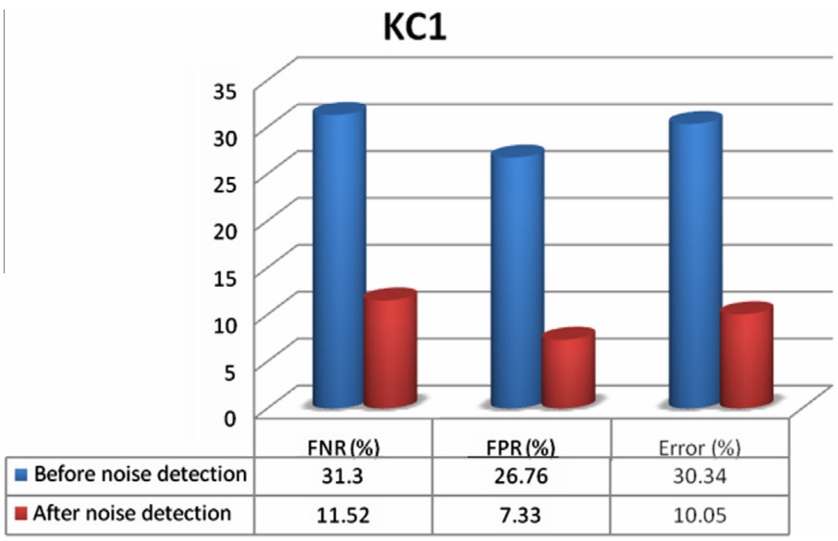
We observed that the FPR, FNR, and error values of the software fault prediction models for all of the datasets decreased after we changed the class labels of noisy instances identified with noise detection approach. Fault prediction models were based on a Naive Bayes machine learning algorithm with a logNum filter because Menzies et al. [19] showed that this approach provided the best performance for the NASA datasets.

4.3. Naive Bayes and logNum filter

The Naive Bayes machine learning algorithm is the best known and most widely used algorithm in data mining. It is not only easy to implement on various kinds of data sets, but it is also quite efficient. Menzies et al. [19] showed that the Naive Bayes algorithm with a logNums filter provides the best performance on the NASA datasets in terms of the software fault prediction problem. The dataset is first transformed into a new dataset by taking the logarithm of the metrics, and then the Naive Bayes algorithm is used on this transformed dataset. Menzies et al. [20] reported that 50 randomly selected modules (i.e., 25 defective and 25 non-defective modules) yielded as much information as larger datasets, and the simple Naive Bayes algorithm [19] again performed as well as anything else. They suggested focusing on the information content of datasets rather than complicated machine learning algorithms. As explained by Menzies et al. [20], a very small sample of datasets provided effective fault predictors, and ignoring large amounts of data was not harmful. We will not give details of the Naive Bayes algorithm in this article due to space limitations, but several papers investigate the different usages of this algorithm [31].

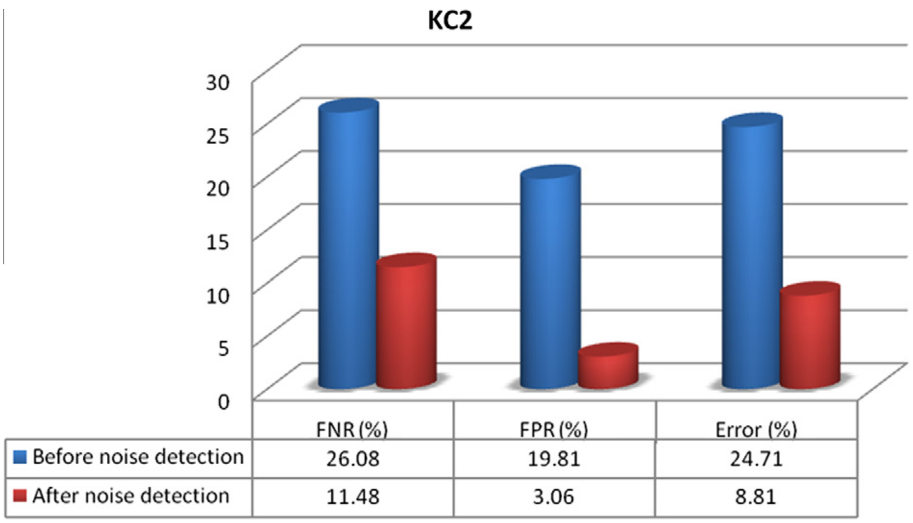
4.4. Results and analysis

First of all, we calculated threshold values for thirteen software metrics for each dataset by using a modified ROC-based approach based on Shatnawi et al. [26]. Tables 3 and 4 show these threshold values. After this step, we identified noisy modules by using a noise detection approach that utilizes software metric threshold values. In order to investigate the effect of the noise detection approach on software fault prediction models, we built prediction models with the original and changed datasets. Only class labels of noisy modules were changed from faulty to non-faulty or non-faulty to faulty. Before applying the Naive Bayes machine learning algorithm on the datasets, we first used a logNum filter on the datasets and then built models with the Naive Bayes algorithm. We observed that the performance of prediction models on the changed datasets was much better than the performance of models on the original datasets. The FPR, FNR, and error rates of the models decreased after the class labels of noisy instances were corrected. This shows that the performance of the fault predictors



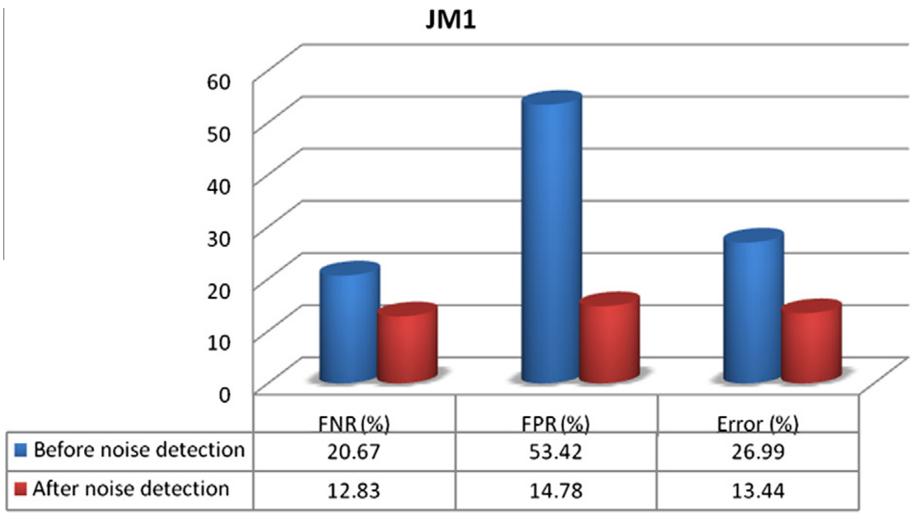
	FNR (%)	FPR (%)	Error (%)
🟦 Before noise detection	31.3	26.76	30.34
🟥 After noise detection	11.52	7.33	10.05

Fig. 2. Performance evaluation parameters for KC1.



	FNR (%)	FPR (%)	Error (%)
🟦 Before noise detection	26.08	19.81	24.71
🟥 After noise detection	11.48	3.06	8.81

Fig. 3. Performance evaluation parameters for KC2.



	FNR (%)	FPR (%)	Error (%)
🟦 Before noise detection	20.67	53.42	26.99
🟥 After noise detection	12.83	14.78	13.44

Fig. 4. Performance evaluation parameters for JM1.

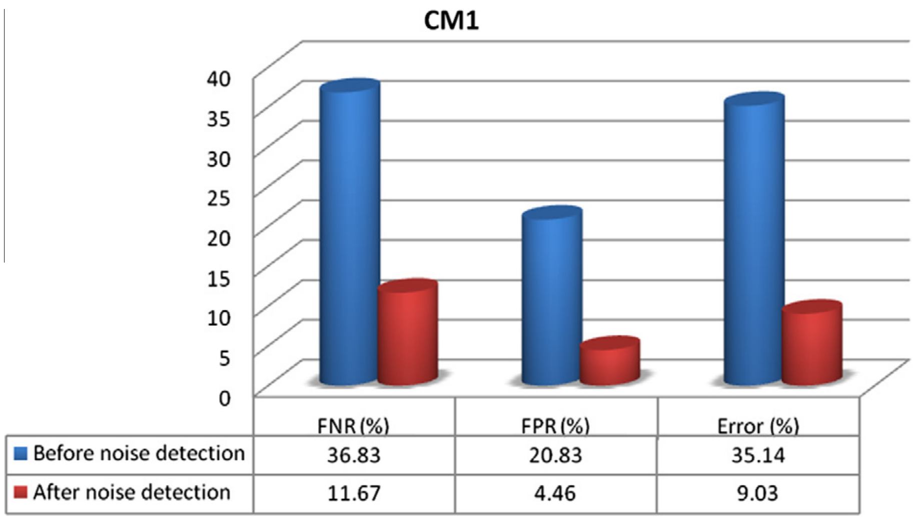
improves when using the Naive Bayes algorithm with a logNum filter if the class labels of identified noisy instances are corrected.

Figs. 2–6 depict the performance variation of the Naive Bayes algorithm-based fault predictors resulting from the use of noise detection approach on the KC1, KC2, JM1, CM1, and PC1 datasets, respectively.

As seen in these figures, the FNR, FPR, and error rates for the models decreased after the noise detection process was performed on all the datasets in experiments. In this study, we calculated metric threshold values depending on the underlying dataset rather than using constant values. Therefore, this approach can be generalized outside the experimental settings and easily integrated into a software project.

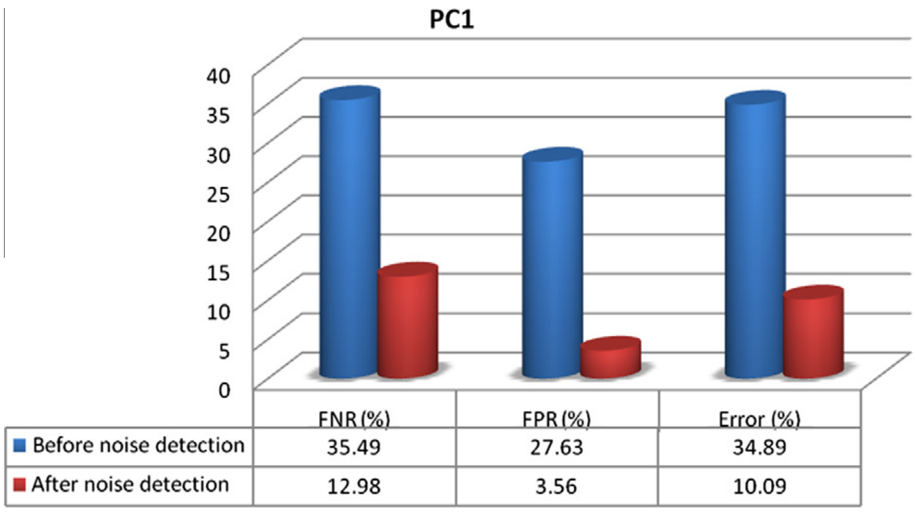
4.5. Validity evaluation

External validity: Threats to external validity limit the generalization of the results outside the experimental setting. A system that can be used in an empirical study must have the following features: "(1) developed by a group, rather than an individual; (2) developed by professionals, rather than students; (3) developed in an industrial environment, rather than an artificial setting; and (4): large enough to be comparable to real industry projects" [14]. Experiments satisfy all of the above-mentioned requirements, and the results can be generalized outside the experimental setting. However, the characteristics of the dataset may affect the performance of noise detection approach.



	FNR (%)	FPR (%)	Error (%)
🟦 Before noise detection	36.83	20.83	35.14
🟥 After noise detection	11.67	4.46	9.03

Fig. 5. Performance evaluation parameters for CM1.



	FNR (%)	FPR (%)	Error (%)
🟦 Before noise detection	35.49	27.63	34.89
🟥 After noise detection	12.98	3.56	10.09

Fig. 6. Performance evaluation parameters for PC1.

Conclusion validity:"Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. Threats to conclusion validity are issues that affect the ability to draw the correct inferences" [14]. In this study, the tests that were used to evaluate the performance of the Naive Bayes-based models were performed using 10-fold cross-validation.
Internal validity: "Internal validity is the ability to show that results obtained were due to the manipulation of the experimental treatment variables" [14]. Experimental treatment variables were not manipulated in this study; instead, they were used as-is. The logNum filter was applied because researchers have reported that the performance of the Naive Bayes-based fault predictors improves when this filter is used.
Construct validity: "This refers to the degree to which the dependent and independent variables in the study measure what they claim to measure" [21]. As we used well-known and previously validated software metrics, we can state that these variables are satisfactory; thus, no threat exists for construct validity. Method-level metrics applied in this study are widely used in software fault prediction studies.

5. Conclusion and future work

Most datamining studies have focused on detecting class noise. Studies showed that class noise impacts classifiers more severely than attribute noise; therefore, the aim was to develop a class noise detection technique in this study. We proposed a noise detection approach that is based on the threshold values of software metrics, and we validated this technique on public NASA datasets. Naive Bayes-based software fault prediction models were built both before and after applying the proposed noise detection algorithm. The FPR, FNR, and error rates of the fault prediction models significantly decreased after the class labels of noisy instances were changed from faulty to non-faulty or non-faulty to faulty. Experimental studies showed that this approach is very effective for detecting noisy instances, and the performance of the Naive Bayes-based software fault prediction model improves when this noise detection approach is used. The main contribution of this study is the proposal of a noise detection technique based on the threshold values of software metrics and extending a metric threshold calculation technique to provide a better decision-making process.

Future work will consider evaluating this noise detection technique on other public datasets located in the PROMISE repository, and we will investigate the threshold calculation technique on large-scale open source projects, such as Eclipse, in order to find the general trend of metric threshold values. Datasets with class-level metrics will be used for further analysis, and the threshold values for object-oriented metrics will be calculated. After this step, we will investigate the effect of noise detection approach on datasets that have class-level metrics.

Acknowledgements

This study is supported by The Scientific and Technological Research Council of Turkey (TUBITAK) under Grant 107E213. The findings and opinions in this study belong solely to the authors, and are not necessarily those of the sponsor.

References

[1] C.E. Brodley, M.A. Friedl, Identifying and Eliminating Mislabeled Training Instances, in: Proc. of the 30th National Conference on Artificial Intelligence, Portland, OR, 1996, pp. 799–805.
[2] X.B. Cao, Y.W. Xu, D. Chen, H. Qiao, Associated evolution of a support vector machine-based classifier for pedestrian detection, Information Sciences 179 (8) (2009) 1070–1077. http://dx.doi.org/10.1016/j.ins.2008.10.020.
[3] V. Chandola, A. Banerjee, V. Kumar, Outlier Detection: A Survey, Technical Report, University of Minnesota, 2007.
[4] K. El Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, S. Rai, The optimal class size for object-oriented software, IEEE Transactions on Software Engineering 28 (5) (2002) 494–509.
[5] K. Erni, C. Lewerentz, Applying design-metrics to object-oriented frameworks, in: Proceedings of the Third IEEE International Symposium on Software Metrics: From Measurement to Empirical Results, Berlin, Germany, 1996, pp. 64–74.
[6] A. Folleco, T.M. Khoshgoftaar, J. Van Hulse, L.A. Bullard, Software quality modeling: The impact of class noise on the random forest classifier, in: Proceedings of the IEEE Congress on Evolutionary Computation, Hong Kong, China, 2008, pp. 3853–3859.
[7] J. Han, M. Kamber, Data Mining: Concepts and Techniques, second ed., Morgan Kaufmann, 2006.
[8] T.M. Khoshgoftaar, L.A. Bullard, K. Gao, Detecting outliers using rule-based modeling for improving cbr-based software quality classification models, in: K.D. Ashley, D.G. Bridge (Eds.), Proceedings of the 16th International Conference on Case-Based Reasoning, vol. 1689, Springer-Verlag, LNAI, Trondheim, Norway, 2003, pp. 216–230.
[9] T.M. Khoshgoftaar, N. Seliya, The necessity of assuring quality in software measurement data, in: Proceedings of the Software Metrics, 10th International Symposium, IEEE Computer Society, Washington, DC, 2004, pp. 119–130. http://dx.doi.org/10.1109/METRICS.2004.41.
[10] T.M. Khoshgoftaar, N. Seliya, K. Gao, Rule-based noise detection for measurement data, in: Proceedings of the Information Reuse and Integration, Las Vegas, Nevada, 2004, pp. 302–307.
[11] T.M. Khoshgoftaar, J. Van Hulse, Empirical case studies in attribute noise detection, IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews 39 (4) (2009) 379–388.
[12] T.M. Khoshgoftaar, S. Zhong, V. Joshi, Enhancing software quality estimation using ensemble-classifier based noise filtering, Intelligent Data Analysis 9 (1) (2005) 3–27.
[13] T.M. Khoshgoftaar, J.V. Hulse, C. Seiffert, A hybrid approach to cleansing software measurement data, in: Proceedings of the 18th IEEE International Conference on Tools with Artificial intelligence, ICTAI, IEEE Computer Society, Washington, DC, 2006, pp. 713–722. http://dx.doi.org/10.1109/ICTAI.2006.11.
[14] T.M. Khoshgoftaar, N. Seliya, N. Sundaresh, An empirical study of predicting software faults with case-based reasoning, Software Quality Journal 14 (2) (2006) 85–111.
[15] D. Li, W. Hu, W. Xiong, J. Yang, Fuzzy relevance vector machine for learning from unbalanced data and noise, Pattern Recognition Letters 29 (9) (2008) 1175–1181. http://dx.doi.org/10.1016/j.patrec.2008.01.009.
[16] G.L. Libralon, A.C.P.L.F. de Carvalho, A.C. Lorena, Pre-processing for noise detection in gene expression classification data, Journal of the Brazilian Computer Society 15 (1) (2009).
[17] J. Liu, Q. Hu, D. Yu, A weighted rough set based method developed for class imbalance learning, Information Sciences 178 (4) (2008) 1235–1256. http://dx.doi.org/10.1016/j.ins.2007.10.002.
[18] E. Menahem, L. Rokach, Y. Elovici, Troika – An improved stacking schema for classification tasks, Information Sciences 179 (24) (2009) 4097–4122.
[19] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Transactions on Software Engineering 33 (1) (2007) 2–13.
[20] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, Y. Jiang, Implications of ceiling effects in defect predictors, in: Proceedings of the 4th International Workshop on Predictor Models in Software Engineering, Leipzig, Germany, 2008, pp. 47–54.
[21] G.J. Pai, J.B. Dugan, Empirical analysis of software fault content and fault proneness using bayesian methods, IEEE Transactions on Software Engineering 33 (10) (2007) 675–686.
[22] T. Quah, Estimating software readiness using predictive models, Information Sciences 179 (4) (2009) 430–445. http://dx.doi.org/10.1016/j.ins.2008.10.005.
[23] L. Rosenberg, Applying and interpreting object oriented metrics, in: Software Technology Conference, Salt Lake City, Utah, 1998.
[24] C. Seiffert, T.M. Khoshgoftaar, J. Van Hulse, A. Folleco, An empirical study of the classification performance of learners on imbalanced and noisy software quality data, in: Proceedings of the Information Reuse and Integration, Las Vegas, Nevada, 2007, pp. 651–658.
[25] N. Seliya, T.M. Khoshgoftaar, S. Zhong, Semi-supervised learning for software quality estimation, in: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence, Boca Raton, FL, 2004, pp. 183–190.
[26] R. Shatnawi, W. Li, J. Swain, T. Newman, Finding software metrics threshold values using ROC curves, Journal of Software Maintenance and Evolution: Research and Practice 22 (1) (2010) 1–16.
[27] K. Ulm, A statistical method for assessing a threshold in epidemiological studies, Statistics in Medicine 10 (3) (1991) 341–349.

[28] J. Van Hulse, T.M. Khoshgoftaar, A comprehensive empirical evaluation of missing value imputation in noisy software measurement data, Journal of Systems and Software 81 (5) (2008) 691–708. http://dx.doi.org/10.1016/j.jss.2007.07.043.
[29] J. Van Hulse, T.M. Khoshgoftaar, Knowledge discovery from imbalanced and noisy data, Data and Knowledge Engineering (2009).
[30] K. Yoon, O. Kwon, D. Bae, An approach to outlier detection of software measurement data using the *K*-means clustering method, in: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, ESEM, IEEE Computer Society, Washington, DC, 2007, pp. 443–445. http://dx.doi.org/10.1109/ESEM.2007.16.
[31] M. Zhang, J.M. Peña, V. Robles, Feature selection for multi-label Naive Bayes classification, Information Sciences 179 (19) (2009) 3218–3229.
[32] S. Zhong, T.M. Khoshgoftaar, N. Seliya, Analyzing software measurement data with clustering techniques, IEEE Intelligent Systems 19 (2) (2004) 20–27. http://dx.doi.org/10.1109/MIS.2004.1274907.
[33] X. Zhu, X. Wu, Class noise vs. attribute noise: a quantitative study of their impacts, Artificial Intelligence Review 22 (3–4) (2004) 177–210.