Expert-Based Software Measurement Data Analysis with Clustering Techniques

Shi Zhong, Taghi M. Khoshgoftaar, and Naeem Seliya
Computer Science and Engineering Department
Florida Atlantic University, Boca Raton, FL 33431, USA
{zhong, taghi, nseliya}@cse.fau.edu

Abstract— Software quality estimation models, used to predict the fault-proneness of software modules based on software metrics, are often constructed by training a classifier from labeled software metrics data. Two challenges often encountered in building an accurate model are the presence of "noisy" data and the possible unavailability of fault-proneness labels in real-world projects. The performance of a model often improves if outliers and noise are removed from the training data. More important, a classifier cannot be trained without fault-proneness labels. This article describes an exploratory analysis method that addresses these two challenges and that is built with clustering and the help of a software engineering expert. It is an unsupervised method since labeled training data are not required to predict the fault-proneness of software modules. We present two real-world case studies to verify the effectiveness of the clustering- and expert-based approach in predicting both the fault-proneness of software modules and potential "noisy" (e.g., mislabeled) modules.

Keywords— **Software Quality Estimation, Exploratory Data Analysis, Clustering, Noise Detection**

I. Introduction

In software quality estimation problems, one typically constructs software quality classification or software fault prediction models using software metrics and fault data (e.g., fault-proneness labels) from a previous system release or similar software project developed previously [1]. Such models are then used to predict the fault-proneness of software modules that are currently under development. This allows for early tracking and detecting of potential software faults, which is critical in many high-assurance telecommunication and medical software systems.

There are two main challenges in building an accurate software quality estimation model: (a) the presence of "noisy" data instances usually degrades trained models; (b) the absence of software quality measurements (fault-proneness labels) renders the construction of a classification model impossible. Clustering is naturally proposed as an exploratory data analysis tool to address these two challenges.

There are generally two types of "noise" in software metrics and quality data. One is related to mislabeled software modules, for which software engineers may either fail to detect, forget to report, or ignore recording any existing software faults. The other has to do with the deficiencies of some of the collected software metrics, which can lead to two similar (in terms of the given metrics) software modules having different fault-proneness labels. It has been shown that removing noisy instances can significantly improve the performance of calibrated software quality estimation models [2]. Therefore, it is desirable to pinpoint the problematic software modules before calibrating any software quality estimation models.

The second challenge is highlighted by the reality of the software development process. In real-world software projects, software fault measurements may not be available for training a software quality estimation model. Such a situation occurs when an organization is dealing with a software project type for which it does not have a similar experience in the past. In addition, software fault data in a previous system release may not have been recorded or collected. Hence the question is: how does the software quality assurance team predict the software quality (of the project under development) based on only the collected software metrics? Under such situations, a supervised learning approach cannot be taken due to the absence of the software quality metric, i.e., risk-based class or number of faults. The estimation task then falls on the analyst (expert) who has to decide the labels for each software module.

Unsupervised learning methods, such as clustering techniques, are a natural choice for analyzing software quality in the absence of fault-proneness labels. Clustering algorithms can group the software modules according to the values of their software metrics. The underlying software engineering assumption is that the fault-prone software modules will have similar software measurements, and hence are likely to be grouped together in the same cluster(s). Similarly, the not fault-prone modules will likely be grouped in the same cluster(s). Upon the completion of clustering analysis, a software engineering expert inspects each different cluster and labels it as either fault-prone or not fault-prone.

A clustering approach for grouping the software modules is of practical benefit to the expert who has to decide the labels of individual software modules. Instead of inspecting and labeling each module one at a time, an expert can inspect and label a given cluster as a whole, i.e., all the modules in the cluster will be assigned the same quality label. Such a strategy leverages upon clustering techniques to group similar modules together as coherent clusters, and eases the tedious (compounded when the number of modules is very large) labeling problem for the expert. For each cluster the clustering algorithm can provide a representative software module, which can be inspected by the expert for labeling all the modules in that cluster (with the aid of other descriptive data statistics).

Moreover, in the availability of the actual labels for the software modules, clustering analysis can provide to the expert valuable feedback for improving the expert-based labeling of software modules of future releases of the given software project or those of other software projects.

The clustering approach to software quality estimation is also attractive for addressing the noise detection and removal problem for software quality classification. It can be a preprocessing step in analyzing the quality of training data prior to building a classifier. For example, for a given cluster if a big majority of software modules are not fault-prone and a few modules are fault-prone, then the expert can make an educated assessment that the few fault-prone modules in the cluster are likely noise and may be eliminated from the data.

TABLE I
Software Metrics.

Category	Metric
Branch Count metric	Branch_Count
Line Count metrics	Total_Lines_of_Code
	Executable_LOC
	Comments_LOC
	Blank_LOC
	Code_And_Comments_LOC
Halstead metrics	Total_Operators
	Total_Operands
	Unique_Operators
	Unique_Operands
McCabe Metrics	Cyclomatic_Complexity
	Essential_Complexity
	Design_Complexity

II. Software Metrics

In software engineering, a large amount of research work has been devoted to developing and studying effective software metrics for reflecting the complexity of a given software system. Software engineering researchers and practitioners alike have proposed several software complexity metrics, most of which use the program code for expressing software complexity. The best known software metrics are Halstead's software metrics and McCabe's complexity metrics. Halstead defined a range of metrics based on the syntactic elements in a program (the operators and operands); McCabe's cyclomatic complexity metric is derived from a program's control flow graph, and is equal to the number of linearly independent executable paths in the program. These software metrics have been widely (and successfully) used even though some issues related to their effectiveness remain as open research problems.

The software measurement datasets used in our experiments contain 13 software metrics, including four Halstead's metrics, three McCabe's complexity metrics, four Line Counts, and a Branch Count, as shown in Table I. Besides the 13 software metrics, a software fault measurement (also called software quality metric or measurement) is available as *Error Rate*, which indicates the number of defects (or software faults) in each software module. It can be used (based on a threshold) to group the software modules into one of two software quality-based groups, i.e., fault-prone and not fault-prone. In our studies the class-membership of a module is also referred to as its *Defect* label.

III. Noise Handling Techniques

In general, there are three different approaches to effectively handling noise in data analysis—designing robust algorithms that are insensitive to noise, filtering out noise [3], and correcting noise [4]. Comparative studies in related research works have shown that which noise handling approach works best depends on specific applications and the given datasets.

Robust algorithms are mostly built with a complexity control mechanism so that the resulting models do not overfit training data and generalize well to future unseen data. Cross validation, Minimum Description Length, and Structural Risk Minimization are some commonly-used model selection principles.

Noise filtering [3] or removal techniques identify and eliminate potential outliers and mislabeled instances in the dataset. One typical machine learning method in this category is to use an ensemble of multiple classifiers and treat the data instances that are misclassified by a given majority of the classifiers as potential data noise and mislabeled instances. In our study, the modules misclassified by the clustering- and expert-based software quality estimation approach are compared to "noise" modules identified by an ensemble of 25 classifiers.

Noise correction [4] or polishing methods are built upon the assumption that each attribute or feature in the data is correlated with others and can be reliably predicted. The correction process starts by predicting the value of each feature for each data instance from other features. Heuristics are then used to determine whether one should change ("correct") the original value of a feature for an instance to the predicted value. This approach is usually more computationally expensive than the first two.

Most of the previous research works have focused on data noise handling algorithms instead of real-world applications, and have used artificially injected noise to evaluate the effectiveness of proposed algorithms. The difficulty with real-world data, as encountered in our experiments, is that there is no ground truth for the identified "noise" instances. We justify the efficacy of our clustering-based approach from different practical view points, including clustering-based labeling accuracy and the benefits to real-world software development; e.g., noise detection and reducing the amount of data needed for decision making.

IV. Clustering Techniques

Generally speaking, clustering techniques can be divided into pairwise clustering and central clustering [5].

Algorithm: k-means clustering
Input: A set of N data vectors $X = \{x_1, ..., x_N\}$ in $\mathbb{R}^d$ and the number of clusters K.
Output: A partition of the data vectors given by the cluster identity vector $Y = \{y_1, ...y_N\}$, $y_n \in \{1, ..., K\}$.
Steps:
1. Initialization: Initialize the cluster centroid vectors $\{\mu_1, ..., \mu_K\}$;
2. Data assignment: For each data vector x_n, set $y_n = \arg\min_k \|x_n - \mu_k\|^2$;
3. Centroid estimation: For cluster k, let $X_k = \{x_n | y_n = k\}$, the centroid is estimated as $\mu_k = \frac{1}{|X_k|}\sum_{x \in X_k} x$;
4. Stop if Y does not change, otherwise go back to Step 2.

Fig. 1. Standard k-means clustering algorithm.

The former, also called similarity-based clustering, groups similar data instances together based on a data-pairwise proximity measure. Graph partitioning-based clustering and single-link hierarchical agglomerative clustering are two commonly used examples in this category. The latter, also called centroid-based or model-based clustering, represents each cluster by a model, i.e., the cluster "centroid". Centroid-based clustering algorithms are often more efficient than similarity-based clustering algorithms. Regular k-means clustering, mixture-of-models clustering, and some more sophisticated variations, belong to this category. We choose centroid-based approaches over similarity-based ones due to efficiency consideration. Similarity-based algorithms usually have a complexity of at least $O(N^2)$ (for computing the data-pairwise proximity measures), where N is the number of data instances. In contrast, centroid-based algorithms are more scalable, with a complexity of $O(NKM)$, where K is the number of clusters and M the number of batch iterations. We study two clustering techniques—k-means and Neural-Gas—for grouping the unlabeled modules of a high-assurance software system.

A. K-means

The widely used standard k-means algorithm is shown in Fig. 1. The objective function it minimizes is given by,

$$MSE = \frac{1}{N}\sum_n \|x_n - \mu_{y_n}\|^2 \ , \tag{1}$$

where $y_n = \arg\min_k \|x_n - \mu_k\|^2$ is the cluster identity of data vector x_n and μ_{y_n} the centroid of cluster y_n. Unless specified otherwise, $\|\cdot\|$ represents L_2 norm. The popularity of the k-means algorithm is largely due to its simplicity, low time complexity, and fast convergence.

B. Neural-Gas (NGas)

The Neural-Gas clustering algorithm [6] is a competitive learning technique with *SoftMax* learning rule. That is, multiple centroids get updated whenever a data instance is visited. The amount of update depends on the distance between the data instance and the cluster centroid. This is in contrast to the *Winner-Take-All* rule used in the k-means algorithm, where only one centroid gets updated every time an instance is visited. It is closely related to the Self-Organizing Map and maximum entropy clustering. In both, a deterministic annealing mechanism is built in the learning/optimization process to avoid shallow minima.

Let x be a data vector, y the cluster index, and μ_y the centroid (mean) of cluster y. The batch version of the Neural-Gas algorithm amounts to iterating between the following two steps:

$$P(y|x) = \frac{e^{-r(x,y)/\beta}}{\sum_{y'} e^{-r(x,y')/\beta}} \ , \tag{2}$$

and

$$\mu_y = \frac{1}{N}\sum_x P(y|x)x \ , \tag{3}$$

where β is an equivalent temperature parameter that controls the smoothness of error surface, and $r(x, y)$ is a rank function that takes the value $k-1$ if y is the kth closest cluster centroid to data vector x. Therefore, there is a rank of clusters relative to each data instance x, and the assignment probability $P(y|x)$ decreases as the rank of cluster y goes down.

Both the k-means and Neural-Gas algorithms have online versions, in which the cluster centroids are updated incrementally when each data vector is visited. The online update rules are available as a gradient decent approach. An online version of the Neural-Gas algorithm has been shown to converge faster and find better local solutions than the Self-Organizing Map and maximum entropy clustering techniques for certain problems [6].

V. Unsupervised Software Quality Estimation

In the context of software quality estimation, most research works have focused on using supervised learning methods for building software quality classification or fault prediction models. There has been very little effort devoted to unsupervised learning for software quality estimation. The most relevant recent work is an unsupervised analysis and visualization of software measurement data using self-organizing maps [7], which did not focus on the software quality estimation or noise detection and filtering problem.

The word "unsupervised" here refers to learning without class labels (i.e., the software fault measurement) and not to learning without human supervision. We emphasize this to avoid some confusion since our analysis involves a software engineering human expert who "supervises" the labeling efforts.

Our clustering- and expert-based software quality estimation is an interactive process. First, depending on the time availability of an expert, we determine a realistic range for number of clusters K. Such an approach is important for software projects where resources are limited. The mean software measurement values of each cluster is then presented to the expert as the cluster representatives. The expert also specifies what other statistics for the software measurement dataset are needed in order to

accurately label each cluster as either fault-prone or not fault-prone. Note that the cluster representatives are not part of a similarity-based clustering process, which further supports the choice of centroid-based clustering.

In our study, the provided data statistics include global mean, minimum, maximum, median, 75 percentile, 80 percentile, 85 percentile, and 90 percentile of each feature (software metric) dimension, as well as the size of each cluster. Of course, in the interactive process, the clustering analyst can suggest other useful information to the software engineering expert, who in turn may ask for additional data properties. In our study, the expert is a professional with over fifteen years of experience in software quality and reliability engineering. Though in this study the software engineering expert is one person, multiple experts (as a realistic scenario in an industrial setting) can function as a team in labeling the software modules based on the clustering information provided.

VI. Experimental Study

A. System Description

The empirical case study presented is on datasets from two NASA software projects (written in C++), which are labeled JM1 and KC2. The software measurements and fault data were collected at the program function, subroutine, or method levels. Hence, a software modules in both case studies is a program function, subroutine, or method. In the JM1 dataset, it was observed that some modules with same attribute values had different *Defect* labels. Upon removing such modules and those with missing values from the JM1 dataset, 8850 modules remained. We label the reduced JM1 dataset as *JM1-8850*. The KC2 dataset has 520 software modules, and is labeled as *KC2-520*.

The collection and use of a given set of software metrics is dependent on the available metrics data collection tools and the software project under consideration. Another software project may consider a different set of software measurements for software quality estimation [8]. Some basic data statistics that are provided to the software engineering expert are shown in Table II and III for JM1-8850 and KC2-520, respectively. They are derived based on interaction with the expert and used by the expert in his/her labeling effort. Each row in the table is one software metric, and the row-order in the table is the same as in Table I.

A majority of the software modules have no faults. For JM1-8850, there are 1687 modules containing at least one fault and as many as 26 faults. For KC2-520, a total of 106 modules have one or more software faults, and the largest number of faults is 13. A module with no defect is labeled as 0, i.e., not fault-prone, while those with one or more faults are labeled as 1, i.e., fault-prone. In our clustering- and expert-based analysis, however, the labels of the modules are not used. The labels are only used for evaluating the expert's decision of which clusters are fault-prone and which are not fault-prone.

TABLE II

Expert-specified data statistics for JM1-8850 dataset.

min	max	mean	median	75%	80%	85%	90%
1	826	11.54	5	13	15	19	25
1	3442	43.73	24	47	57	71	94
0	2824	32.04	17	34	41	52	69
0	344	3.36	0	3	4	6	9
0	447	5.66	3	6	8	9	13
0	108	0.45	0	0	0	0	1
1	5420	83.25	40	85	104	134	180
0	3021	56.73	27	59	72	92	126
1	411	13.40	12	17	18	20	22
0	1026	20.35	14	24	27	33	41
1	470	6.49	3	7	8	10	13
1	165	3.33	1	3	4	5	7
1	402	4.14	2	4	5	6	8

TABLE III

Expert-specified data statistics for KC2-520 dataset.

min	max	mean	median	75%	80%	85%	90%
1	361	8.79	3	9	11	15	19
1	1275	37.03	13	45	55	64	91
0	1107	27.87	8	34	42	52	71
0	44	2.0	0	2	2	3	5
0	121	4.35	1	5	7	9	11
0	11	0.28	0	0	0	0	1
1	2469	57.83	17	64	80	106	142
0	1513	37.16	11	41	53	70	95
1	47	9.23	8	14	15	16	18
0	325	14.52	7	20	24	28	37
1	180	4.91	2	5	6	8	10
1	125	2.45	1	1	1	4	5
1	143	3.66	2	4	5	6	7

B. Experimental Setting

Both the k-means and Neural-Gas algorithms are implemented in C, with an interface to Matlab and compiled into mex files that can be called directly from Matlab. The executables are in binary code thus run faster than regular Matlab code. Batch training was used for the k-means algorithm with a relative convergence criterion of *1e-4* (i.e., if the relative change of cost function between two consecutive iterations is less than *1e-4*). For the datasets we have, the k-means algorithm always converges within 100 iterations. We used an online version of the Neural-Gas algorithm and the temperature parameter β starts at $\frac{K}{2}$ and gradually decreases to 0.01 at the end of 100 training epochs.

The number of clusters K (a parameter needed by both the k-means and Neural-Gas algorithms) is set to 30 for JM1-8850 and 20 for KC2-520. The heuristic consideration for choosing 20 and 30 is a balance between reducing the number of software module representatives to be examined by the expert and obtaining a fine (granularity) representation of the original software measurement data. Both the k-means and Neural-Gas algorithms generated a few empty clusters (4 and 2, respectively) for KC2-520; thus the actual number of clusters evaluated by the expert is less than 20 for KC2-520. Nevertheless, instead of evaluating 520 or 8850 software modules one by one, the

expert only needs to label less than 20 or 30 groups based on the respective cluster centroids, other data statistics, and his software engineering knowledge.

For clustering quality, we use the mean squared error (*mse*) objective (Equation 1) and average purity (*ave-pur*). The purity of a cluster is defined as the percentage of the most dominated category (fault-prone or not fault-prone) in the cluster, and average purity is the mean over all clusters. It has a range of between 0 and 1; the higher the number, the better is the average purity.

Defect labels provided with the dataset are used to evaluate the expert's labeling decision, but are not used in the expert-based labeling process. Specifically, the overall classification error, false positive rate (*fpr*, percentage of not fault-prone modules labeled as fault-prone by the expert), and false negative rate (*fnr*, percentage of fault-prone modules labeled as not fault-prone) are reported.

C. Clustering and Quality Estimation Results

The clustering quality results are shown in Table IV. The Neural-Gas algorithm performs significantly better in terms of *mse* and comparably in terms of average purity. The k-means algorithm, however, runs much faster. The run time results are recorded on a 3.06GHz Pentium 4 PC running Windows XP.

TABLE IV

CLUSTERING RESULTS WITH 30 CLUSTERS FOR *JM1-8850* AND 20 CLUSTERS FOR *KC2-520*.

		mse	*ave_pur*	*time (seconds)*
JM1-8850	k-means	3342.7	0.746	0.966
	NGas	1574.4	0.727	9.928
KC2-520	k-means	738.3	0.808	0.016
	NGas	244.0	0.815	0.375

Figure 2 reports the overall classification error, false positive rate, and false negative rate results, for both k-means and Neural-Gas algorithms, compared to a decision tree-based classifier (C4.5). The C4.5 classifier is chosen for comparison because it is a commonly used and known for its robustness in classification accuracy. Since for a given classification technique, the *fpr* and *fnr* are inversely related, the C4.5 classification results were obtained by adjusting the parameters (such as tree-depth, pruning ratio, etc.) in C4.5 to achieve *fpr* errors similar to the two clustering techniques.

The Neural-Gas algorithm performs slightly worse for JM1-8850 and only slightly better for KC2-520, even though in Table IV it gives significantly lower *mse* value. One possible reason for this is noise contained in the datasets (wrong labels or insufficient attributes). The error numbers are very comparable to the C4.5 results, suggesting that the expert-based software quality classification is a viable option (compared to supervised learning) when the software quality metric data is not available.

One point worth mentioning is that classification for the JM1-8850 dataset is very difficult even for many state-of-the-art classifiers. For example, using the *libsvm* software package (available from http://www.csie.ntu.edu.tw/∼cjlin/libsvm/) with a two-fold cross validation setting, the support vector machine method achieves only 20% overall accuracy, with a false positive rate of 0 but a false negative rate of 98%. That is, the support vector machine method classifies almost all data as not fault-prone. A similar observation with supervised classification was also made for the KC2-520 dataset. However, the difficulty in classification for the KC2-520 dataset was relatively lower (as seen in Figure 2) than the JM1-8850 dataset. The promising accuracy results in Figure 2 warrant more future investigations in building clustering- and expert-based software quality prediction systems.

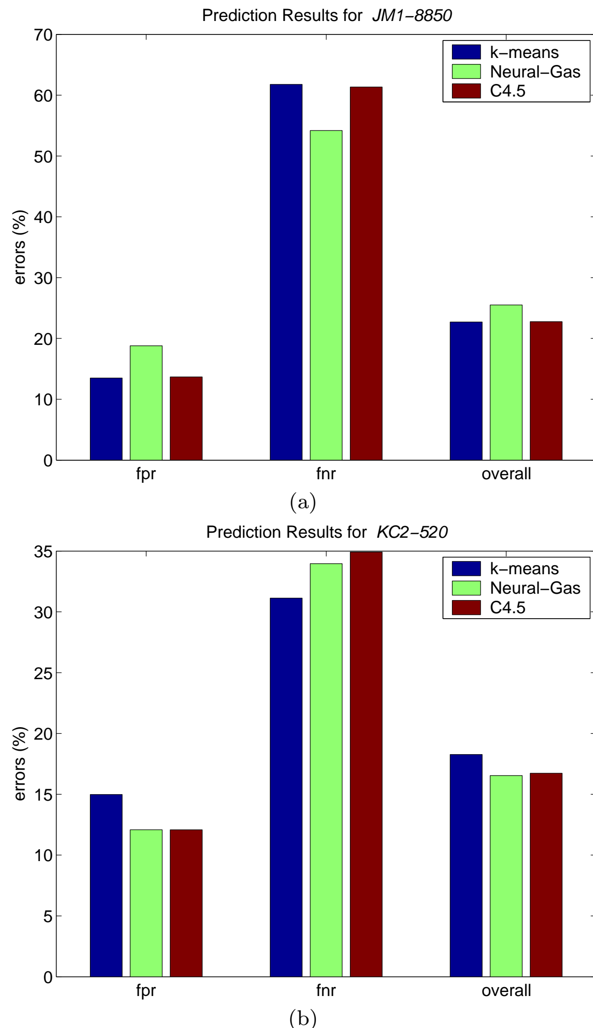

Fig. 2. Clustering- and Expert-based classification results on (a) JM1-8850 dataset; (b) KC2-520 dataset.

A feedback from the software engineering expert shows that the Neural-Gas results seem to be easier to label than the k-means results. We suspect the reason is that the Neural-Gas algorithm generates more coherent clusters. This is important for a real interactive data analysis system since the expert will have higher confidence explaining the clusters derived from the software metrics

data.

D. Classification Models for Ensemble Noise Filter

In a recent study [9], we performed some preliminary work on detecting noise on the two NASA software case studies using an ensemble of 25 classifiers. The basic idea in that study was to use a set of different classifiers (as an ensemble-classifier filter) to classify the same dataset and tag, as potential noise, those software modules that are misclassified by a majority number of classifiers. The underlying assumption is that if a large number of classifiers misclassify a given software module, then it is likely that it's software attribute values do not adhere to the underlying characteristics of the software product as a whole, and hence, the software module is likely noise.

TABLE V
Classifiers Used for Ensemble Noise Filter

Symbol	Classifier
CBR	Case-Based Reasoning [1]
TD	Treedisc Decision Tree Algorithm
LR	Logistic Regression
LOC	Lines of Code
GP	Genetic Programming
ANN	Artificial Neural Networks
LBOOST	LogitBoost Classifier
RBM	Rule-Based Modeling [2]
BAG	Bagging Classifier
RSET	Rough Sets-Based Classifier
MCOST	MetaCost Classifier
ABOOST	AdaBoost Classifier
DTABLE	Decision Table
ADT	Alternating Decision Table
SMO	Sequential Minimal Optimization
IB1	Instance-Based (1 nearest neighbor)
IBK	Instance-Based (k nearest neighbor)
PART	Partial Decision Trees
ONER	OneR Algorithm (based on one most informative attribute)
JRIP	Repeated Incremental Pruning to Produce Error Reduction
RDR	Ripple Down Rule Algorithm
J48	C4.5 Decision Tree Algorithm
NBAYES	Naive Bayes
HPIPES	Hyperpipes Algorithm
LWLS	Locally Weighted Learning

The ensemble noise filter consists of 25 different classifiers, which included: traditional classifiers such as logistic regression, naive Bayes, and regression trees; and advanced computational intelligence-based methods such as genetic programming, neural networks, and rough sets. The 25 classifiers used in our ensemble noise filter are listed in Table V. The classifiers encompass different supervised learning paradigms, including induction, soft-computing, and statistical regression. Several of the classifiers are implemented in the WEKA data mining and machine learning tool, which is written in Java [10]. More details for each classifier can be found in [9] and [10].

For each classification technique, a number of candidate classification models were first obtained. The one with a preferred balance between the *fpr* and *fnr* error rates was chosen as the final model. Such a strategy was adopted in accordance to our previous studies [1] of high assurance software systems that are similar to JM1 and KC2. Moreover, such a model-selection strategy provides a practical solution to the problem of having very few fault-prone modules in the software dataset of a high-assurance system, where supervised classification is difficult and often yields a classifier predicting most (or all) modules as not fault-prone (the majority class).

In a Lines of Code (LOC) classifier, the modules were first sorted in an ascending order of their lines of code. Based on a specific threshold value of lines of code, thd_{LOC}, the modules with LOC lower than thd_{LOC} are then selected and labeled (predicted) as not fault-prone, and the rest as fault-prone. The threshold value is varied until the desired balance between the *fpr* and *fnr* error rates is obtained. The LOC metric is often used in software engineering practice as a rule-of-thumb to gauge the quality of a software product. The underlying assumption is that a larger-size program module is likely to have more software faults than a relatively smaller-size module.

The *fpr* and *fnr* error rates of the 25 classifiers for the two case studies are shown in Table VI. In most of the classifiers (19 out of 25), the error rates are based on a 10-fold cross-validation approach. For the six classification techniques marked with an '$*$' in the table, a cross-validation feature was not available. The last five rows of the table present some descriptive statistics of the error rates for the 25 classification models. We observe that the error rates of the selected models for the different classifiers are relatively greater for the JM-8850 dataset than the KC2-520 dataset. In addition, we observe that the *fpr* and *fnr* error rates for a given classifier are similar, reflecting our model-selection strategy.

E. Noise Detection Results

We compare the "noise" modules tagged by the ensemble noise filter approach with those mislabeled by the software engineering expert in the clustering-based method. The results in Figure 3 show an interesting match between the two sets of modules. The x-axis indicates the level of consensus among the 25 classifiers used for noise-filtering. For example, 13C means that a module is viewed as "noise" if 13 or more classifiers (out of 25) in the ensemble predict the label wrong. The y-axis shows the recall percentage of modules considered as "noise" by the ensemble, i.e., how many of them are covered by the set of mislabeled modules by the expert.

In the case of the JM1-8850 dataset, the noise recall performance of the expert-based classification with clusters obtained by Neural-Gas are generally better than those based on k-means. In the case of the KC2-dataset, how-

TABLE VI
Error Rates for JM1-8850 and KC2-520

Methods	JM1-8850		KC2-520	
	fpr	*fnr*	*fpr*	*fnr*
CBR	30.70%	30.88%	21.50%	20.75%
TD *	30.78%	29.16%	18.60%	16.04%
LR *	34.23%	33.97%	20.77%	21.70%
LOC *	34.85%	34.08%	20.53%	19.81%
GP *	34.71%	32.66%	18.36%	16.98%
ANN *	38.06%	30.35%	21.26%	21.70%
LBOOST	34.72%	32.72%	22.22%	20.75%
RBM	33.71%	33.08%	17.39%	16.04%
BAG	30.59%	30.76%	21.50%	20.75%
RSET *	31.62%	30.94%	16.18%	14.15%
MCOST	33.67%	33.61%	23.43%	21.70%
ABOOST	33.41%	33.79%	28.26%	29.25%
DTABLE	34.29%	34.32%	18.84%	18.87%
ADT	33.83%	33.61%	19.81%	19.81%
SMO	34.09%	33.97%	20.77%	20.75%
IB1	34.73%	34.74%	23.67%	24.53%
IBK	32.70%	32.48%	20.53%	19.81%
PART	33.16%	33.14%	20.77%	19.81%
ONER	34.50%	34.38%	20.05%	19.81%
JRIP	33.18%	33.08%	19.81%	19.81%
RDR	33.94%	34.02%	18.84%	19.81%
J48	32.56%	32.42%	19.57%	19.81%
NBAYES	34.12%	33.97%	21.26%	21.70%
HPIPES	37.97%	38.29%	23.91%	23.58%
LWLS	33.59%	33.61%	20.05%	19.81%
Average	33.75%	33.12%	20.71%	20.30%
Std. Dev	1.80%	1.80%	2.41%	2.93%
Median	33.83%	33.61%	20.53%	19.81%
Min	30.59%	29.16%	16.18%	14.15%
Max	38.06%	38.29%	28.26%	29.25%

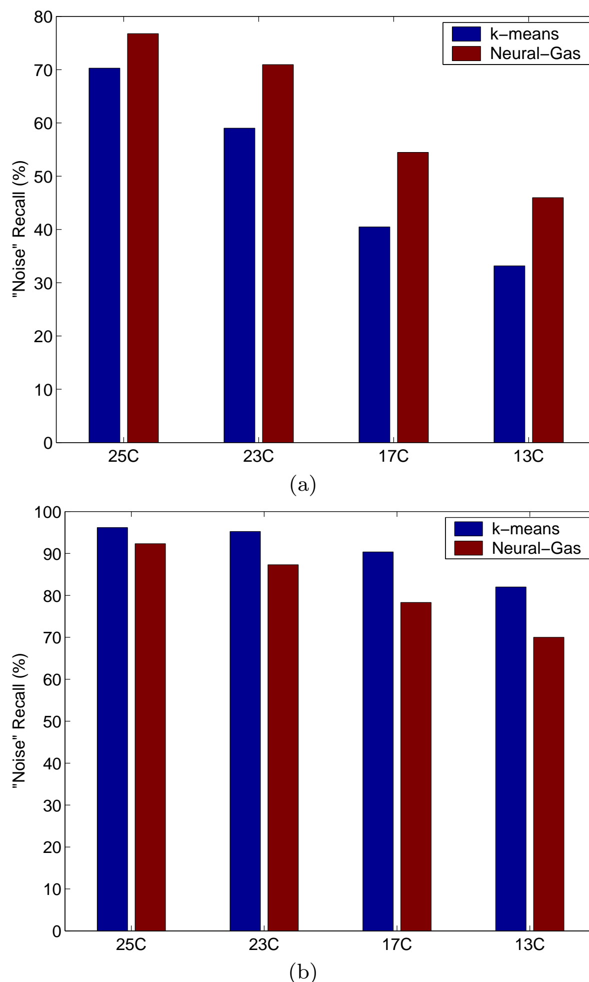

Fig. 3. "Noise" recall results on (a) *JM1-8850* dataset; (b) *KC2-520* dataset.

ever, the noise recall of classification based on k-means is generally better than those based on Neural-Gas. The absolute noise recall performance of the expert-based classification is generally better for the KC2-520 dataset than the JM1-8850 dataset, suggesting that performance of a classifier is influenced (among other factors) by the characteristics of the dataset including the extent of potential noise in the data.

It is interesting to notice that a majority of the modules detected as "noise" are among the modules mislabeled by the clustering- and expert-based labeling method. Even though currently we do not know which one (the noise-filtering method or the clustering method) is more accurate for this case study, the important point here is that the interesting matches warrant future research on noise filtering with unsupervised clustering techniques.

VII. Outlook

This study reflects our initial research in exploring clustering-based analysis for software quality estimation problems. We plan to exchange further discussions with software engineers to better evaluate the benefits of our clustering-based analysis. Further detailed interpretation on the quality estimation and "noise" detection results is also needed for this purpose.

A more interactive system can be built for software engineers to explore software metrics data, identify mislabeled software modules, and pinpoint the deficiency and inappropriateness of collected software metrics. With closer collaboration between data analysts and software engineering experts, more informative software metrics may be constructed and collected.

An expert-based classification scheme using clustering techniques can be applied for classification problems of other domains, such as medical research and computer network intrusion detection. Future work will consider additional clustering techniques, and compare the performances with those obtained in this study. In addition, the impact of the numbers of clusters presented to the expert on the obtained classification accuracy deserve more investigation.

References

[1] Taghi M. Khoshgoftaar and Naeem Seliya, "Analogy-based practical classification rules for software quality estimation," *Empirical Software Engineering Journal*, vol. 8, no. 4, pp. 325–350, December 2003.

[2] Taghi M. Khoshgoftaar, Lofton A. Bullard, and Kehan Gao, "Detecting outliers using rule-based modeling for improving cbr-based software quality classification models," in *Proc. 5th Int. Conf. on Case-Based Reasoning*, K. D. Ashley and D. G. Bridge, Eds. 2003, vol. LNAI 1689, pp. 216–230, Springer-Verlag.

[3] Carla E. Brodley and Mark A. Friedl, "Identifying mislabeled training data," *Journal of Artificial Intelligence Research*, vol. 11, pp. 131–167, 1999.

[4] C. M. Teng, "Correcting noisy data," in *Proceedings of the Sixteenth International Conference on Machine Learning*, 1999, pp. 239–248.

[5] Shi Zhong and Joydeep Ghosh, "A unified framework for model-based clustering," *Journal of Machine Learning Research*, vol. 4, pp. 1001–1037, November 2003.

[6] T. M. Martinetz, S. G. Berkovich, and K. J. Schulten, ""Neural-Gas" network for vector quantization and its application to time-series prediction," *IEEE Trans. Neural Networks*, vol. 4, no. 4, pp. 558–569, July 1993.

[7] Witold Pedrycz, G. Succi, Marc Reformat, P. Musilek, and X. Bai, "Self organizing maps as a tool for software analysis," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, Toronto, Canada, May 2001, IEEE Computer Society, vol. 1, pp. 93–97.

[8] L. C. Briand, W. L. Melo, and J. Wust, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 706–720, July 2002.

[9] Vedang Joshi, "Noise elimination with ensemble-classifier filtering: A case study in software quality engineering," M.S. thesis, Florida Atlantic University, Boca Raton, Florida USA, December 2003, Advised by Taghi M. Khoshgoftaar.

[10] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2000.