Dealing with Noise in Defect Prediction

Sunghun Kim[1], Hongyu Zhang[2], Rongxin Wu[2] and Liang Gong[2]

[1] Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong
[2] School of Software, Tsinghua University, Beijing, China

hunkim@cse.ust.hk, hongyu@tsinghua.edu.cn, {se.wu.rongxin, jacksongl1988}@gmail.com

ABSTRACT

Many software defect prediction models have been built using historical defect data obtained by mining software repositories (MSR). Recent studies have discovered that data so collected contain noises because current defect collection practices are based on optional bug fix keywords or bug report links in change logs. Automatically collected defect data based on the change logs could include noises.

This paper proposes approaches to deal with the noise in defect data. First, we measure the impact of noise on defect prediction models and provide guidelines for acceptable noise level. We measure noise resistant ability of two well-known defect prediction algorithms and find that in general, for large defect datasets, adding FP (false positive) or FN (false negative) noises alone does not lead to substantial performance differences. However, the prediction performance decreases significantly when the dataset contains 20%-35% of both FP and FN noises. Second, we propose a noise detection and elimination algorithm to address this problem. Our empirical study shows that our algorithm can identify noisy instances with reasonable accuracy. In addition, after eliminating the noises using our algorithm, defect prediction accuracy is improved.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement–*Restructuring, reverse engineering, and reengineering,* D.2.8 [**Software Engineering**]: Metrics – *Product metrics,* K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software maintenance*

General Terms

Algorithms, Measurement, Experimentation

Keywords

Defect prediction, noise resistance, buggy changes, buggy files, data quality.

1. INTRODUCTION

Defect prediction is a very active area in software engineering research [7, 10, 11, 13, 18, 19, 33, 35]. Many effective new metrics and algorithms to predict defect-proneness have been proposed. When researchers evaluate their new algorithms or metrics, they often use defect information collected from the change logs in Software Configuration Management (SCM) systems and from the bug reports in bug tracking systems.

Unfortunately, recent studies have found that extracted defect information from change logs and bug reports are noisy. For example, Aranda and Venolia et al. [1] manually inspected ten bug reports in Microsoft and interviewed developers related to the reports. They found lots of important information missing in bug reports. Bird et al. [4] also studied the quality of change logs and bug reports, and found that many change logs and bug reports were not linked. They also found that the noisy defect information could seriously affect the performance of a bug prediction algorithm.

These surprising findings challenge the validity of all existing bug prediction algorithms by raising important questions: How could we deal with the noise in the defect data? Are existing defect prediction algorithms still useful if their prediction models are trained by noisy defect data? How much noise is acceptable for bug prediction algorithms? How could we detect and eliminate the noise?

This paper addresses these questions. First, we propose a method, which intentionally adds false positive and negative information only in the training data to measure noise resistance of a given bug prediction algorithm. Using the proposed method, we measure noise resistance of two well-known bug prediction algorithms, change classification and buggy file prediction. We found that these two algorithms are relatively noise resistant. When there are enough buggy instances in the datasets, defect prediction performance (measured in terms of F-measure) does not decrease significantly with the increases of false positive or false negative noises. We also find that these algorithms are more resistant to false negative noises. However, the prediction performance decreases significantly when the dataset contains 20%-35% of both FP and FN noises.

Second, we propose an algorithm to detect and eliminate noises in the defect data to address the noisy data problem. We experimentally evaluate our algorithm and the results show that it can identify noisy instances with reasonable accuracy. In addition, after eliminating the noises using our algorithm, the defect prediction accuracy is improved.

Overall, this paper makes the following contributions:

- **Noise resistance measuring technique**: We propose a method to measure noise resistance of defect prediction models.
- **Empirical study of measuring noise resistance**: We apply the resistance measuring method for two well-known prediction algorithms, and provide guidelines for acceptable noise level.

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

- **Noise detection technique:** We propose an accurate noise detection algorithm, which also improves defect prediction accuracy.

In the remainder of the paper, we start by presenting the background on defect prediction algorithms in Section 2. In Section 3, we discuss the noisy defect data issue. We propose a noise resistance measuring method in Section 4 and apply it for change classification and buggy file prediction in Section 5. We present our noise detection algorithm in Section 6. Section 7 discusses the threats to validity. We round off the paper with related work in Section 8 and conclusions in Section 9.

2. BACKGROUND

2.1 A General Defect Prediction Process

Before measuring noise resistance of defect prediction algorithms, we describe a common defect prediction process as shown in Figure 1. Then we introduce two well-known defect prediction algorithms used in this paper.

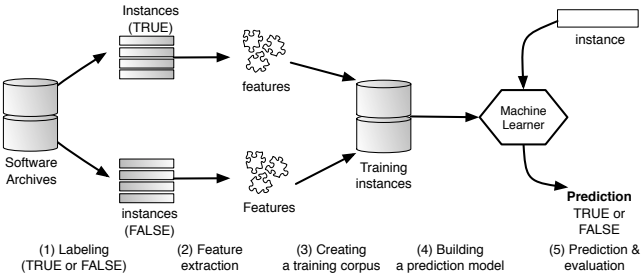


Figure 1. A general defect prediction process

Before designing a prediction model, we need to specify the prediction target. A prediction model can be used for predicting defect-proneness (buggy or clean) of different software entities, such as a component [18, 32], file [16, 19, 33] or a change [9, 11]. After deciding the prediction target, a general defect prediction process (Figure 1) can be as follows:

Labeling: Defect data need to be collected for training a prediction model. This process typically involves extracting instances (data items) from software archives and labeling them as TRUE (buggy) or FALSE (clean). However, some recent studies have discovered that data collected by mining software repositories often contain noise. Noisy data threaten the validity of prediction models. We will describe this issue in Section 3.

Extracting features and creating training corpus: This step extracts features for predicting the labels of instances. Common features for defect prediction are complexity metrics, keywords, changes, and structural dependencies. By combining labels and features of instances, we can create a training corpus to be used by a machine learner to construct a prediction model.

Building prediction models: Using a training corpus, general machine learners such as Support Vector Machines (SVM) or Bayes Net can be used to build a prediction model. The model can then take a new instance and predict its label, i.e. TRUE or FALSE.

Evaluation: To evaluate a prediction model, we need a testing data set besides a training set. We predict the labels of instances in the testing set and evaluate the prediction model by comparing the prediction and real labels. To separate the training and testing sets, 10-fold cross-validation is widely used.

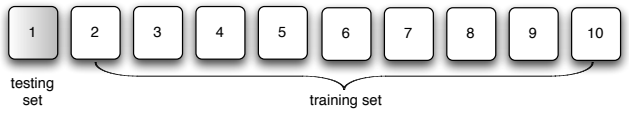


Figure 2. 10-fold cross-validation

In 10-fold cross-validation, the data is divided equally into 10 folds as shown in Figure 2. Then the instances in each fold are in turn used as a testing set and the remaining nine folds are used to train the model. For example, in the first iteration, instances in *fold2* to *fold10* are used as a training set, and *fold1* as a testing set.

There are four possible outcomes from a prediction model: classifying a buggy instance as buggy ($n_{b\to b}$), classifying a buggy instance as clean ($n_{b\to c}$), classifying a clean instance as clean ($n_{c\to c}$), and classifying a clean instance as buggy ($n_{c\to b}$). The recall, precision, and F-measures are widely used to evaluate prediction results [27, 31]. We use these measures to evaluate prediction models as follows:

- Precision (buggy) = $\frac{n_{b\to b}}{n_{b\to b} + n_{c\to b}}$

  This is the number of correct classifications of the type ($n_{b\to b}$) over the total number of predicted buggy instances.
- Recall (buggy) = $\frac{n_{b\to b}}{n_{b\to b} + n_{b\to c}}$

  This is the number of correct classifications of the type ($n_{b\to b}$) over the total number of actual buggy instances.
- F-measure (buggy) = $\frac{2 * P(b) * R(b)}{P(b) + R(b)}$

  This is a composite measure of precision and recall. We use the F1 metric that weights recall and precision equally [27].

2.2 Software Defect Prediction Algorithms

In this section, we describe two well-known defect prediction models used to measure noise resistance.

2.2.1 Predicting Buggy Changes

Change Classification (CC) learns buggy change patterns from history, and predicts if a new change introduces bugs or not [9,11].

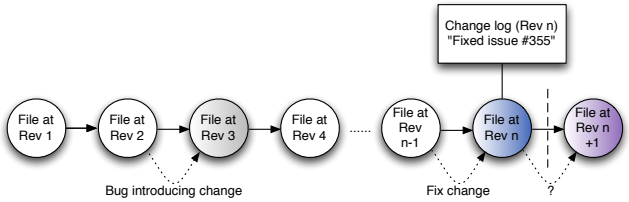


Figure 3. Change history with buggy and fix changes

Suppose we have a change history of a file as shown in Figure 3. After learning from buggy and clean change patterns from revision 1 to revision *n*, CC predicts if the change in revision *n+1* introduces bugs.

To learn from history, CC extracts features from each change. To label changes as buggy or clean changes, first we need to extract changes from project history. Then we identify fix changes using change logs. To extract change history, we use Kenyon [3], a system that extracts source code change histories from SCM systems such as CVS and Subversion. Kenyon automatically checks out the source code of each revision and extracts change

information such as the change log, author, change date, source code, and change deltas.

Once a commit has been determined to contain a fix, it is possible to trace backward in the revision history to determine when the fixed erroneous code is introduced in the system. We define that as a bug-introducing change. The bug-introducing change identification algorithms proposed by Śliwerski et al. [23] and Kim et al. [11] are used.

A file change involves two source code revisions (an old revision and a new revision) and a change delta that records the added code (added delta) and the deleted code (deleted delta) between the two revisions. A file change has associated metadata, including the change log, author, and commit date. By mining change histories, we can derive features such as co-change counts to indicate how many files are changed together in a commit, the number of authors of a file, and the previous change count of a file. Every term in the source code, change delta, and change log texts is used as features. Detailed feature extraction methods of CC can be found in [11].

2.2.2 Predicting Buggy Files

Another common defect prediction model is identifying buggy files in advance. It is widely believed that some internal properties of software (e.g., metrics) have relationships with external properties (e.g., defects). In recent years, many defect prediction models based on software metrics have been proposed (e.g., [13, 16, 18, 19, 33]). These prediction models identify code features (expressed as measurement data), learn a classification model from historical defect data, and use the constructed model to predict defect-proneness of a new program module.

Many code features can be extracted from software projects to predict defective files. These features include complexity metrics (such as lines of code, cyclomatic complexity, number of classes, etc.), process metrics (such as the number of lines of code changes, the number of file changes, etc.) and resource metrics (such as developer information, etc). All these metrics, or a combination of these metrics, can be used to build effective software defect prediction models.

3. NOISES IN DEFECT DATA

Both prediction algorithms described in Section 2 require labels (buggy or clean) to build and evaluate models. In this section, we discuss typical techniques to identify labels and the noise in the labels.

To label a file/change as buggy or clean, many researchers mine the bug database and version achieves for open source systems. Two approaches are widely used: searching for keywords such as "Fixed" or "Bug" [14] and searching for references to bug reports like “#42233” [23] . We use both techniques in our experiments. Chen et al. [5] studied open source change log quality. They checked the correctness of each change log and found almost all logs were correct.

Some open source projects have strong guidelines for writing their change logs. For example, 100% of Columba’s change logs used in our experiment have a tag such as ‘[bug]’, ‘[intern]’, ‘[feature]’, and ‘[ui]’. Usually, Eclipse developers leave relevant bug report IDs in their change logs.

However, some recent studies (such as those reported by Bird et al. [4]) discovered that data collected via mining software repositories (MSR) often contain noise. They found that the number of linked bugs (bugs whose change logs and bug reports are linked) does not match the number of total fixed bugs (the ratio could be even lower than 50%), suggesting a high percentage of false negatives in the defect dataset. This is because developers often do not write specific keywords or leave links for fix revisions. It is also possible that developers make mistakes when they write keywords or links in the change logs. For this reason, automatically collected defect data based on these keywords or links are inevitably noisy. Recent studies have also found that noisy data (in training and testing sets) affect performance of prediction models [4].

We also performed a replication study of Bird et al.’s experiments. Our results confirm their findings about noisy defect data. For example, for the Eclipse SWT component, there are 32% unlinked bugs (bugs that do not reflected in CVS logs) in Eclipse 3.0 and 21% unlinked bugs in Eclipse 3.1. The existence of the unlinked bugs indicates that the defect data collected via MSR is noisy. We also noticed that the noise level decreased in Eclipse 3.4, where 92.27% SWT bugs are recorded in CVS logs. In this paper, we measure the effect of noise on two defect prediction models described in Section 2, and propose an algorithm to detect the noise in Section 6.

4. EXPERIMENTAL SETUP

4.1 Research Questions

Our experiments are designed to address the following research questions:

RQ1: How resistant a defect prediction model is to false negative (FN) buggy data?

RQ2: How resistant a defect prediction model is to false positive (FP) buggy data?

RQ3: How resistant a defect prediction model is to both false negative (FN) and false positive (FP) buggy data?

As Bird et al. [4] found out, developers often forget to leave explicit messages or links to indicate buggy changes. Since most automatic buggy change/file identifications are based on special keywords and links [11, 16, 28, 33, 36] in the change logs, this will lead to false negatives (missing some buggy changes) in the automatically identified data. RQ1 measures predictor resistance for this case.

On the other hand, it is possible that developers label a change/file as buggy by leaving special keywords and bug report links, together with some non bug-fix changes in one commit. This behavior leads to false positives (identifying non-buggy changes/files as buggy). RQ2 measures resistance of defect prediction models to false positives in the training data set.

Finally, RQ3 measures the noise resistant ability of defect prediction models when data has both false positives and false negatives.

4.2 Making Noisy Data

To address the research questions, we first need a *golden set*, which contains no FPs and FNs. In addition, we need noisy data sets. However, it is very hard to get a golden set. In our approach, we carefully select high quality datasets and assume them the golden sets. Then, to create noise sets, we add FPs and FNs intentionally into the golden sets. To add FPs and FNs, we

randomly selects instances in a golden set and artificially change their labels from buggy to clean or from clean to buggy, inspired by experiments in [4].

Figure 4. Creating biased training set

To make FN data sets (for RQ1), we randomly select *n%* buggy labeled instances and change their labels to clean, as shown in Figure 4 (1). Similarly, to make FP data sets (for RQ2), we select *n%* of clean labeled instances and change their labels to buggy, which adds false buggy changes, as shown in Figure 4 (2). For the FN and FP data sets (for RQ3), we select random *n%* of instances, and change their labels. For example, if a clean-labeled instance is selected, we change its label to buggy. If a buggy instance is selected, we change its label to clean.

It is very important to note that we add noise only in the training set, not in the testing set. For testing, we use the original golden set. In this way, we can measure the accuracy of a defect prediction model, which is trained from noisy data sets, to predict buggy/clean changes in the golden set.

In this paper, we use the 10-fold cross validation described in Section 2. First, we group 9 folds to be used as a training set. Then, we add noise only in the training set and leave the testing set unchanged.

For the machine learner, we use the Bayes Net classifier (the Weka implementation [26]). Bayesian networks have good performance when dealing with a large number of variables with much variance in values [27]. We also compare performances of other machine learners in Section 5.3.2.

4.3 Dummy Predictor

An effective defect prediction model should outperform at least random guessing – guessing a change/file as buggy or clean purely at random. We call a predictor based on random guessing a *dummy predictor*. Since there are only two labels, buggy and clean changes, the dummy predictor could also achieve certain prediction accuracy. For example, if there are 30% buggy changes in a project, by predicting all changes as buggy, the buggy recall would be 1 and the precision would be 0.3. It is also possible that the dummy predictor randomly predicts a change as buggy or clean with 0.5 probability. In this case, the buggy recall would be 0.5, but still the precision is 0.3.

We use the F-measure of the dummy predictor as a reference line when measuring the noise resistance of defect prediction models. We compute the dummy F-measure assuming the dummy predictor randomly predicts 50% as buggy and 50% as clean. For example, for a project with 30% buggy changes, the dummy buggy F-measure is 0.375 $\left(2 \times \frac{0.5 \times 0.30}{0.5 + 0.30}\right)$.

5. NOISE RESISTANCE

This section reports our experiments on the impact of noise on two defect prediction algorithms and discusses the results.

5.1 Noise Resistance of Change Classification

5.1.1 Subject Programs

We use Columba, Eclipse JDT.Core and Scarab as our subjects for this experiment (Table 1), as these projects have high quality change logs and links between changes logs and bug reports. For the first two projects, we adopt the exact datasets used in [11], which were also used by other researchers [2, 21]. We assume these datasets as golden sets and use them to measure noise resistance.

5.1.2 Original Accuracy

First, we build a CC prediction model using the original training set and measure the performance of the model using a testing set. Figure 5 shows the buggy recall, prediction and F-measure. Overall, the accuracy results for the first two projects are comparable to those reported in [11] (the small variations in results coming from the use of Bayes Net instead of SVM and the randomness in the 10-fold cross-validation.) For Columba, the buggy precision and recall are around 0.5 to 0.55. For Eclipse, the buggy recall is 0.88, and precision is 0.48. We notice that the precision for Eclipse reported in [11] is 0.61, which is higher than our precision, 0.48. However, our recall is 0.88, which is much higher than the recall 0.61 reported in [11]. This happens due to the recall-precision tradeoff. To address this issue, we use F-measure [26] to measure the noise resistance of CC in this paper.

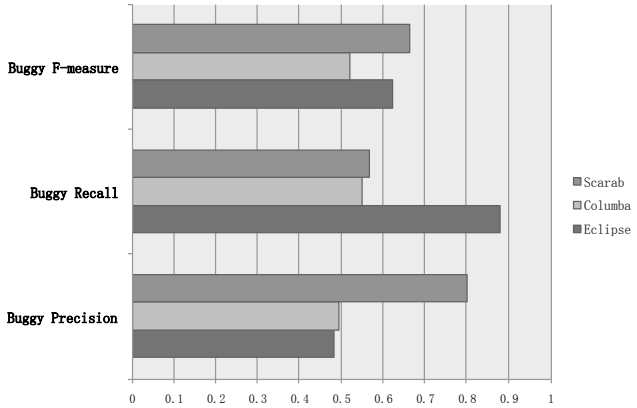


Figure 5. Defect prediction using the original training set

Table 1. Analyzed subject programs for predicting buggy changes

Project	Revisions	Period	# of clean instances	# of buggy instances	% of buggy instances	# of features
Columba	500-1000	05/2003-09/2003	1,270	530	29.4	17,411
Eclipse	500-750	10/2001-11/2001	592	67	10.1	16,192
Scarab	500-1000	06/2001-08/2001	724	366	50.6	5,710

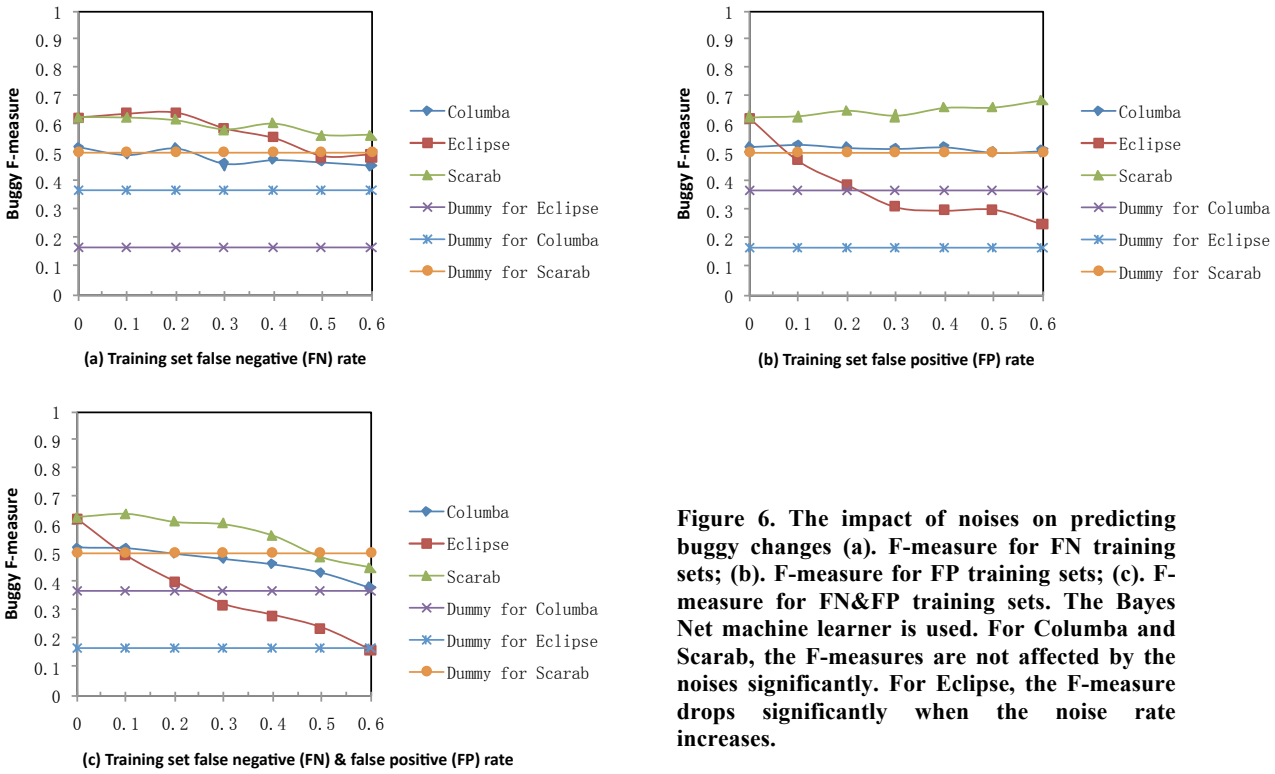


Figure 6. The impact of noises on predicting buggy changes (a). F-measure for FN training sets; (b). F-measure for FP training sets; (c). F-measure for FN&FP training sets. The Bayes Net machine learner is used. For Columba and Scarab, the F-measures are not affected by the noises significantly. For Eclipse, the F-measure drops significantly when the noise rate increases.

5.1.3 FN Resistance (RQ1)

In this section, we measure the resistance of CC for false negative (FN) training sets. To add FNs, we randomly select buggy instances in the training set and label them as clean as shown in Figure 4 (1). In this way, we increase the rate of FN by changing buggy labels to clean. For example, suppose we have 100 buggy instances in a training set. Changing labels of 10 buggy instances to clean will add 10% FN.

Figure 6 (a) shows buggy F-measure results for Columba, Eclipse and Scarab with various FN training sets. The x-axis indicates the FN rates. The dummy F-measures described in Section 4.3 are also shown in the Figures as the reference lines.

For Columba, the buggy F-measure shows strong resistance against the FN training sets. The F-measure values are relatively stable. When the noises reach 60%, the F-measure just drops 0.05. The same can be observed for Scarab, the buggy F-measure is not affected by FN noises significantly. After 20% false negatives are injected into the training set, the F-measure is not changed. When the noises reach 60%, the F-measure only drops less than 0.05.

For Eclipse, the buggy F-measure is just slightly affected by the FN training sets too. After adding 40% FN noises to the training set, the F-measure drops from 0.62 to 0.55. When the noises reach 60%, the F-measure still remains at 0.50.

A possible explanation of these results is that the features characterizing bugs are often common across the buggy changes. Therefore losing some instances in the training set does not lead to significant performance decrease.

5.1.4 FP Resistance (RQ2)

We also observe CC F-measures using FP training sets. We add FPs into the training sets as described in Figure 4 (2) and then perform change classifications. The results are shown in Figure 6 (b).

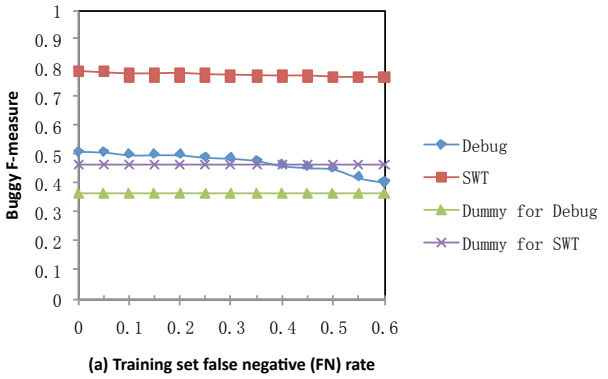
For Columba and Scarab, the buggy F-measures are not significantly affected by the false positives. For Eclipse, buggy F-measures are affected by the FN training sets. After adding 20% FP noises to the training set, the F-measure drops from 0.6 to 0.4. After having more than 50% FP noises, the F-measure is close to that of the dummy predictor.

A possible explanation of the sensitivity of the Eclipse F-measures is the small number of buggy changes in the dataset. There are only 67 buggy changes as shown in Table 1. After adding many FPs, the features that characterize bugs become less obvious for classifiers to learn. On the other hand, Columba and Scarab all have more than 300 buggy changes to learn from, the features characterizing bugs can be still identified and prediction performance is still kept.

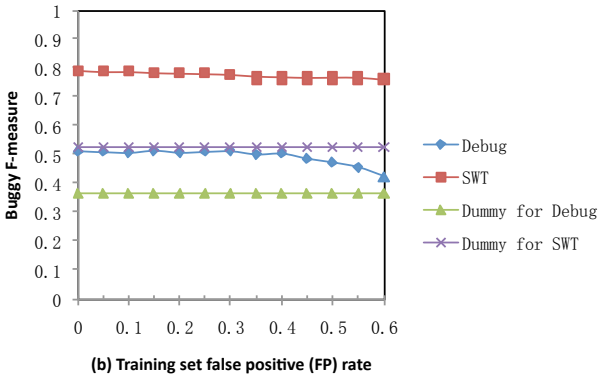
5.1.5 FN and FP Resistance (RQ3)

We also examine the prediction performance when the training sets contain both FP and FN noises. As shown in Figure 6 (c), the trend of buggy F-measures for all projects decline when the noise rate increases.

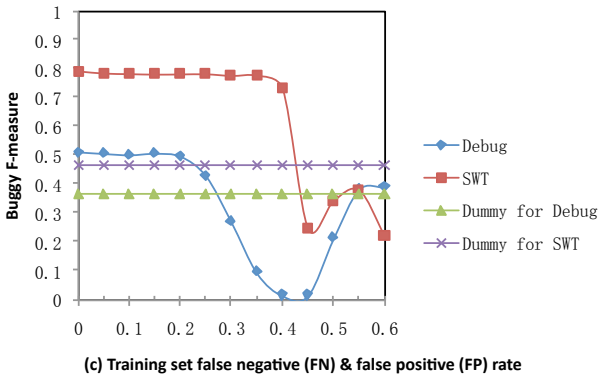
For Columba and Scarab, their F-measures only decrease by 0.1-0.15 when noise level reaches 60%. Interestingly, the F-measure of Eclipse decreases much faster than that of Columba and Scarab. Note that, Columba and Scarab have many buggy/clean instances and switching some labels dose not hurt the prediction too much. However, the F-measure of Eclipse significantly drops when the level of FP and FN noise increases. After the noises reach 40%, the Eclipse's F-measures are almost the same as the dummy F-measures.



(a) Training set false negative (FN) rate



(b) Training set false positive (FP) rate



(c) Training set false negative (FN) & false positive (FP) rate

Figure 7. The impact of noises on predicting buggy files (a). F-measure for FN training sets; (b). F-measure for FP training sets; (c). F-measure for FN&FP training sets. The Bayes Net machine learner is used. The F-measures of the Debug and SWT projects are not affected by the FN or FP noises significantly. However, when FN&FP noises reach certain level, the F-measures drop significantly.

5.2 Buggy File Prediction

5.2.1 Subject Programs

To obtain the "golden set" for building prediction models for buggy-files, we use the SWT and Debug projects in Eclipse 3.4. We collected the defect data by mining the Eclipse Bugzilla and CVS repositories. We find that both projects have a high percentage of linked bugs (bugs whose changes logs and bug reports are linked). For SWT, 92.27% bugs reported in Bugzilla are linked to changes. For Debug, 95.92% bugs are linked. Therefore, we use these two datasets as the golden sets.

Table 2 summarizes the datasets used in this study. The SWT dataset contains 1,485 Java source files, among which 43.9% files are defective. The Debug dataset contains 1,065 files, among which 24.69% are defective. We have also collected the following metrics for each file in the projects. These metrics capture different aspects of a file and are used as features for constructing our defect prediction model:

- **Complexity metrics**: including LOC (lines of code), average cyclomatic complexity measure, maximum cyclomatic complexity measure.
- **Object-oriented metrics**: including the WMC, CBO, NOC, DIT, LCOM, RFC metrics that are proposed by Chidamber and Kemerer [6].
- **Change metrics**: including the number of added and deleted lines of code since the last major revision, the number of times the file is changed.
- **Developer metric**: the number of developers who changed the file.

Following the method described in Section 4.2, we intentionally make the dataset noisy by randomly selecting a given percentage of instances and changing their class labels (buggy or clean), thus artificially creating false positives and false negatives. We again use 10-fold cross validation to evaluate the prediction results. We first randomly partition the whole dataset into 10 folds. We use 9 folds as a training set and inject noise into them, and then use the remaining unchanged 1 fold as the testing set. The Bayes Net classifier is used to construct the prediction model.

Table 2. The dataset used for predicting buggy files

Project	LOC	#programs (src files)	#defects	#defective programs	% of linked bugs
SWT	386K	1485	556	653(43.97%)	92.27%
Debug	77K	1065	294	263(24.69%)	95.92%

5.2.2 FN Resistance (RQ1)

Figure 7(a) shows how FN (false negative) training sets affect prediction performance. Clearly, the defect prediction model has strong resistance against the FN training sets. For SWT, the buggy f-measure using the original dataset is 0.79. With the increases of noises, the prediction results are still very stable (with f-measures around 0.78), even when the false negative rate reaches 60%. Similar results are found for the Debug project, which exhibits stable performance until the FN rate reaches 50%. Although some buggy instances are marked as clean, the remaining buggy instances can capture the program features and can be still used for training prediction models effectively.

5.2.3 FP Resistance (RQ2)

Figure 7(b) shows how FP (false positive) training sets affect prediction performance. Similarly, the defect prediction model has

resistance against the FP training sets. The prediction results are very stable. For SWT, the F-measure values are all around 0.78 even the FP rate is 60%. For Debug, the F-measures are about 0.50 until the FP rate reaches 50%. The data noise introduced by false positives does not decrease the prediction accuracy significantly.

5.2.4 FN and FP Resistance (RQ3)

A training set may contain both false positives and false negatives. Figure 7(c) shows how FN and FP noises affect prediction accuracy. For SWT, once the FN and FP noise rate reaches 40%, prediction accuracy starts decreasing quickly. For Debug, prediction accuracy drops after the FN and FP noise rate exceeds 20%. These results show that FN and FP noises together have larger impact on defect prediction.

5.3 Discussions

5.3.1 Acceptable Noise Rate

Many bug prediction approaches use software history to build prediction models. It is often very difficult to collect perfect historical datasets that have no FPs and FNs. How much noise is acceptable for prediction approaches?

Our experiments show that CC and the buggy file prediction yield reasonably stable accuracy at the presence of noises, when the Bayes Net learner is used. When the number of buggy instances is large enough, increasing FP or FN noises does not affect prediction performance significantly. For datasets with both FP and FN noises, the prediction performance decreases when the noises increase. When the number of buggy instances in a dataset is small, the prediction performance will be affected by noises significantly.

In defect prediction practices, FNs are more common as some defects recorded in bug tracking systems are not linked to CVS/SVN logs [4]. FPs happen when developers leave a message saying he fixed a bug, but he actually did not. Chen et al. [5] studied the correctness of open source change logs, and they find that when developers leave a message indicating fixing of bugs, it is likely a real fix. Our experimental results show that noises in FN or FP alone do not affect prediction performance significantly. Also, up to 20%-35% of FP and FN noises (together) usually do not affect the performance significantly either.

Obviously, our results may not be generalizable to all prediction models, but at least these can serve as guidelines for CC and the buggy file prediction users. We suggest that before using these predictors, users can sample their data and manually inspect them to measure FP and FN rates. Based on the rates, they can decide if their defect data is applicable for these predictors.

5.3.2 Noise Resistances of Different Machine Learners

In previous sections, we obtained our results using the Bayes Net machine learner. In this section, we use Naïve Bayes, Support Vector Machines (SVM) and Bagging learners [26, 27] to repeat the experiments and observe the impact of data noises on prediction accuracy.

Figure 8 shows the noise resistance ability of the four machine learners under different False Negative rates. Similar to Bayes Net, the Naïve Bayes learner also has strong noise resistance ability when predicting buggy files. The F-measures do not change significantly when FN rates are increasing. All Bayesian classifiers are based on the Bayer's rule. The classifier is interested in the most probable hypothesis. Therefore, even if there is a certain amount of noise in the defect dataset, which could affect calculation of probability for some hypothesis, the Bayesian classifiers can still make correct classifications when the most probable hypothesis is preserved.

The SVM learner performs poorly with noisy data – F-measures decrease quickly when FNs increase, until FN rate reaches 50%. SVM performs classification by constructing an N-dimensional hyperplane that optimally separates the data into two categories. The noise in the data could affect the construction of the hyperplane considerably. Therefore more noise could lead to more bias in classification.

The Bagging (Bootstrap Aggregating) classifier is a machine learning algorithm that ensembles meta-algorithms to build models. In this experiment, we use the Multilayer Perceptron algorithm as the meta-algorithm [26]. The Bagging classifier separates a training set into several new training sets by random sampling, and builds models based on the new training sets. The final classification result is obtained by the voting of each model. Figure 8 shows that Bagging can improve the original prediction performance, and resist a certain amount of noises. However, when the noise level exceeds 40%, the probability of each model making a wrong classification is increasing, causing the quick drop of the performance.

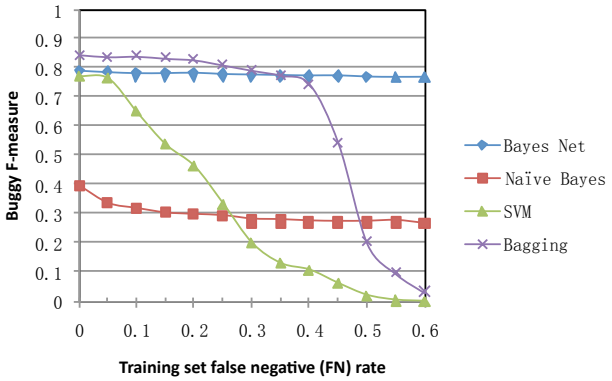


Figure 8. SWT defect prediction results of different machine learners (F-measures for FN training set).

6. HANDLING NOISES IN DEFECT DATA

This section proposes a noise detection algorithm and presents its evaluation.

6.1 Identifying Noisy Instances

We investigate possible methods for identifying noisy instances in defect datasets. If we can detect noises in advance, it is possible to eliminate them and make the data more suitable for predictors.

We propose a novel noise detection algorithm, called Closest List Noise Identification (CLNI). The pseudo-code of the algorithm is given in Figure 9.

The CLNI algorithm works as follows. In each iteration j, for each instance $Inst_i$, its closest instances are listed; we call it $List_i$. In $List_i$ the instances are sorted in ascending order according to their Euclidean Distance to $Inst_i$. The percentage of top N instances that have different class values from $Inst_i$ is recorded as θ. If θ is more

than or equal to a given threshold δ, then $Inst_i$ is highly probable to be a noisy instance and will be included in noise set A_j. The above process is repeated until the similarity between A_j and A_{j-1} is over ε. A_j will be returned as the identified noise set. Empirical study found that when N is 5, δ is 0.6 and ε is 0.99, this algorithm performs the best.

CLNI Algorithm:

```
for each iteration j
    for each instance Inst_i
      for each instance Inst_k
        if(Inst_k ∈ A_{j-1})
          continue;
        else
          add EuclideanDistance(Inst_i, Inst_k) to List_i ;
        end
      end
      calculate percentage of top N instances in List_i
        whose label is different from Inst_i as θ;
      if θ ≥ δ
        A_j = A_j ∪ Inst_i;
      end
    end
    if |A_j∩A_{j-1}| / Max(|A_j|, |A_{j-1}|)≥ ε
      break;
    end
end
return A_j
```

Figure 9. The pseudo-code of the CLNI algorithm

The high-level idea of CLNI can be illustrated as in Figure 10. The blue points represent clean instances and the white points represent buggy instances. When checking if an instance I is noisy, CLNI first lists all instances that are close to I (the points included in the circle). CLNI then calculates the ratio of instances in the list that have a class label different from that of I (the number of white points over the total number of points in the circle). If the ratio reaches a specific threshold δ, we consider instance I to have a high probability to be a noisy instance.

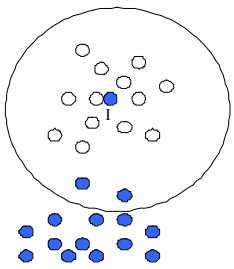


Figure 10. An illustration of the CLNI algorithm

6.2 Evaluation

We evaluate CLNI using data from the Eclipse 3.4 SWT and Debug projects as described in Section 5.2. These two datasets are considered as the golden sets as most of their bugs are linked bugs. Following the method described in Section 4.2, we create the noisy datasets for these two projects by selecting random $n\%$ of instances and artificially changing their labels (from buggy to clean and from clean to buggy). We then apply the CLNI algorithm to detect noisy instances that we have just injected. We use Precision, Recall and F-measures to evaluate the performance in identifying the noisy instances.

Table 3 shows the results when the noise rate is 20%. The Precisions are above 0.6, Recalls are above 0.83 and F-measures are above 0.71. These promising results confirm that the proposed CLNI algorithm is capable of identifying noisy instances.

Table 3.The performance of CLNI in identifying noisy instances

	Precision	Recall	F-measure
Debug	0.681	0.871	0.764
SWT	0.624	0.830	0.712

Figure 11 also shows the performance of CLNI under different noise levels for the SWT component. When the noise rate is below 25%, F-measures increase with the increase of the noise rates. When the noise rate is above 35%, CLNI will have bias toward incorrect instances, causing F-measures to decrease.

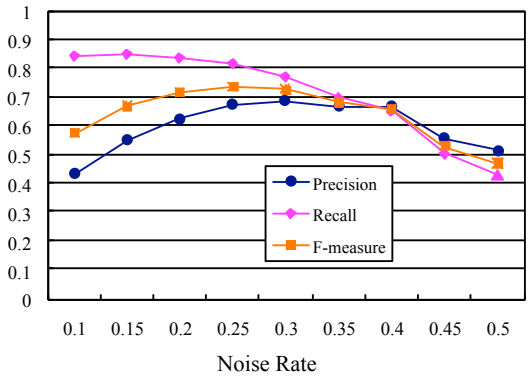


Figure 11. Performance of CLNI with different noise rates

After identifying the noises in the noisy Eclipse 3.4 SWT and Debug datasets using CLNI, we eliminate these noises by flipping their labels. We then evaluate if the noise-removed training set improves prediction accuracy.

The results for the SWT component before and after removing FN and FP noises are shown in Table 4. In general, after removing the noises, the prediction performance (F-measure) improves for all learners, especially for those that do not have strong noise resistance ability. For example, for the SVM learner, when 30% FN&FP noises were injected into the SWT dataset the F-measure was 0.339. After identifying and removing the noises, the F-measure jumped to 0.706. These results confirm that the proposed CLNI algorithm can improve defect prediction performance for noisy datasets.

Table 4.The defect prediction performance (F-measure) after identifying and removing noisy instances (SWT)

Remove Noises ?	Noise Rate	Bayes Net	Naïve Bayes	SVM	Bagging
No	15%	0.781	0.305	0.594	0.841
	30%	0.777	0.308	0.339	0.781
	45%	0.249	0.374	0.353	0.350
Yes	15%	0.793	0.429	0.797	0.838
	30%	0.802	0.364	0.706	0.803
	45%	0.762	0.418	0.235	0.505

7. THREATS TO VALIDITY

We note some threats to the validity of this work.

- *All datasets used in our experiments are collected from open source projects.* The types of noises introduced by open source developers may be different from those introduced by employees in a well-managed software organization. We need to evaluate if commercial projects also exhibit similar noise resistance behavior in defect prediction. This remains as future work.
- *The golden set used in this paper may not be perfect.* For example, there are still a few percentages of bugs that are not linked to the CVS logs. Even though some files are annotated with bug IDs, they may not be the files that actually contain the bugs. It is also possible that a few bugs may not even be recorded in the bug tracking system. Our results may be under threat if the golden sets contain a large number of FPs and FNs.
- *The noisy data simulations used in our experiment may not reflect the actual noise patterns in practice.* In our experiments, instances to be included as FP/FN training sets are randomly selected. It is possible that in practice, occurrences of some noises actually follow certain patterns; for example, files developed by a poorly managed team are more likely to contain noisy defect data.

8. RELATED WORK

8.1 The Data Quality Problem

Real-world data are often noisy, which may affect interpretations and models derived from the data. The data quality problem is well recognized in the data mining area. Some studies show that errors in a large dataset are common and field error rates are typically around 5% or more [29, 12]. Many existing learning algorithms have integrated various approaches to handle noises. For example, the well-known Decision Tree algorithm uses tree-pruning methods to avoid over fitting problems introduced by noises in training data [20]. Zhu and Wu [30] descried a quantitative study of the impact of noisy data on classification accuracies using the UCI machine learning datasets. They found that although some machine learning algorithms have been designed to accommodate noises, noises in class labels can still lower classification accuracies. They also suggest preprocessing methods (such as eliminating instances containing class noise) to enhance classification accuracy.

The data quality problem has also been observed by some software engineering researchers. For example, Mockus [15] noted that in many realistic scenarios the data quality is low (e.g., some change data could be missing), which could affect the outcome of an empirical study. He proposed to use multiple imputation methods to mitigate the effects of missing values. Myrtveit et al. [17] and Strike et al. [22] also noticed the problem of missing and incomplete data in software effort estimation. In this paper, we address the problem of noisy data in software defect prediction.

8.2 The Quality of Software Defect Data

Research on software defect prediction has received much attention in recent years, as the ability to predict defect-proneness of a software module is important for software quality improvement and project management. Many defect prediction models have been proposed (e.g., [7, 10, 11, 13, 16, 17, 35]). However, almost all defect prediction models do not take noise in the data into consideration.

As described in Section 3, many current defect prediction models are built based on data collected by mining software repositories (MSR). Bird et al. [4] reported that the data collected in this manner could introduce a large amount of noises. Although they have noticed the noisy defect data problem, they did not empirically measure the impact of different noise levels on defect prediction accuracy or try to eliminate noise. The noisy data problem does not pertain to data collected by MSR only. It may occur in industrial metric projects as well. For example, Khoshgoftaar and Seliya [8] performed an extensive study on NASA MDP datasets. They observed low prediction performance and suggested that "instead of focusing on searching for another classification technique for improving prediction accuracy, the quality of the software measurement data should be addressed". They also proposed a noise elimination technique based on the k-means algorithm [25]. They detected outliers in the data and treated them as noisy instances. The limitation of their method is that mislabeled instances are often not outliers. In this paper, we present one of the first empirical studies of the impact of noisy data on defect prediction. We also propose a novel noise detection algorithm, which can identify mislabeled instances with good accuracy.

9. CONCLUSIONS

Defect data collected based on specific bug fix keywords or bug report links in change logs are commonly used to build defect prediction models and to evaluate the models. Since leaving specific keywords or bug report links in change logs is optional, automatically collected defect data from change logs inevitably includes noise. Recent studies show that noise in defect data is not negligible, and this noise affects prediction performance [4]. However, the issue of dealing with noisy data has not been addressed adequately.

In this paper, we have introduced a method to measure noise resistance in software defect prediction (for predicting buggy files and buggy changes). By applying the method to two well-known defect prediction models, we found that in general, noises in the defect data do not affect defect prediction performance in a significant manner. However, the prediction performance decreases significantly when the dataset contains 20%-35% of both FPs and FNs.

We have also proposed a new method called CLNI for identifying noisy instances in defect data. Our experiment results show that CLNI can effectively identify noises with reasonable accuracy. The noise-eliminated training sets produced by CLNI can improve the defect prediction performance, especially for the machine learners that do not have strong noise resistant ability.

In future, we will further investigate techniques for improving defect prediction accuracy under noisy environment. We will also explore if the results obtained in this paper are applicable to industrial projects.

All data used in our experiments are available at:
http://code.google.com/p/hunkim/wiki/HandlingNoise

ACKNOWLEDGEMENTS

This research is supported by the Chinese NSF grant 61073006 and the 2010 Microsoft SEIF Awards.

REFERENCES

[1] J. Aranda and G. Venolia, The secret life of bugs: Going past the errors and omissions in software repositories. *Proc. ICSE'09,* Vancouver, Canada, May 2009, 298-308.

[2] L. Aversano, L. Cerulo, and C. Del Grosso, Learning from bug-introducing changes to prevent fault prone code. In *Ninth International Workshop on Principles of Software Evolution (IWPSE'07)*. Dubrovnik, Croatia, Sep 2007.

[3] J. Bevan, E. Whitehead Jr., S Kim and M. Godfrey, Facilitating Software Evolution with Kenyon, *Proc. ESEC/FSE'05*, Lisbon, Portugal, 2005, 177-186.

[4] C. Bird, A. Bachmann, E.Aune, J. Duffy, A. Bernstein, V.Filkov, and P. Devanbu, Fair and balanced?: bias in bug-fix datasets. *Proc. ESEC/FSE '09*, August 2009, 121-130.

[5] K. Chen, S. R. Schach, L. Yu, J. Offutt and G. Z. Heller, Open-Source Change Logs, *Empirical Software Engineering,* vol. 9, September 2004, 197 – 210.

[6] S.R. Chidamber and C.F. Kemerer, A Metrics Suite for Object-Oriented Design, *IEEE Trans. Software Eng.*, vol. 20, 476-493, 1994.

[7] A. E. Hassan, Predicting Faults Using the Complexity of Code Changes, *Proc. ICSE'09*, Vancouver, Canada, May 2009.

[8] T.M. Khoshgoftaar and N. Seliya, The Necessity of Assuring Quality in Software Measurement Data, *Proc.10th Int'l Symp. Software Metrics (METRICS'04)*,119-130, 2004.

[9] S. Kim, T. Zimmermann, K. Pan and E. Whitehead Jr., Automatic Identification of Bug-Introducing Changes, *Proc. ASE'06*, Tokyo, Japan, September 2006.

[10] S. Kim, T. Zimmermann, E. Whitehead Jr., A. Zeller, Predicting Faults from Cached History, *Proc. ICSE'07*, Minneapolis, USA, 2007

[11] S. Kim, E. Whitehead Jr. and Y. Zhang, Classifying Software Changes: Clean or Buggy?, *IEEE Trans. of Software Engineering* vol. 34, 181-196, March/April 2008.

[12] J. Maletic and A. Marcus, Data Cleansing: Beyond Integrity Analysis. *Proc. the Conference on Information Quality (IQ2000)*, 2000.

[13] T. Menzies, J. Greenwald and A. Frank, Data Mining Static Code Attributes to Learn Defect Predictors, *IEEE Trans. Software Engineering,* 32(11), 1-12, 2007.

[14] A. Mockus and L. G. Votta, Identifying Reasons for Software Changes Using Historic Databases, *Proc. 16th International Conference on Software Maintenance (ICSM 2000)*, San Jose, CA, USA, 2000, 120-130.

[15] A. Mockus. Missing Data in Software Engineering. In F.Shull et al. (eds.), *Guide to Advanced Empirical Software Engineering*, 185-200, 2008.

[16] R. Moser, W. Pedrycz and G. Succi, A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction, *Proc. ICSE'08*, Leipzig, Germany, May 2008.

[17] I. Myrtveit, E. Stensrud, and U. H. Olsson. Analyzing Data Sets with Missing Data: An Empirical Evaluation of Imputation Methods and Likelihood-Based Methods. *IEEE Trans. on Software Engineering,* 27(11), 999-1013, 2001.

[18] N. Nagappan, T. Ball,and A. Zeller, Mining Metrics to Predict Component Failures, *Proc. ICSE'06*, Shanghai, China, May 2006.

[19] T. Ostrand, E. Weyuker and R. Bell, Predicting the Location and Number of Faults in Large Software Systems, *IEEE Trans. Software Engineering*, 31 (4), 340-355, 2005.

[20] J.R. Quinlan, Learning from Noisy Data. *Proceedings of the Second International Machine Learning Workshop*, University of Illinois at Urbana-Champaign, 1983.

[21] S. Shivaji, E. Whitehead Jr., R. Akella and S. Kim, Reducing Features to Improve Bug Prediction. *Proc. ASE'09,* Nov 2009. 600-604.

[22] K. Strike, K. E. Emam, and N. Madhavji. Software Cost Estimation with Incomplete Data. *IEEE Trans. on Software Engineering*, 27(10), 890-908, 2001.

[23] J. Śliwerski, T. Zimmermann and A. Zeller*,* When Do Changes Induce Fixes?, in *Int'l Workshop on Mining Software Repositories (MSR 2005)*, Saint Louis, Missouri, USA, 2005, 24-28.

[24] J.Spacco, D. Hovemeyer and W. Pugh, Tracking Defect Warnings Across Versions, in *Int'l Workshop on Mining Software Repositories (MSR 2006)*, Shanghai, China, 2006.

[25] W. Tang and T.M. Khoshgoftaar, Noise identification with the k-means algorithm, *Proc. 16th IEEE Int'l Conference on Tools with Artificial Intelligence (ICTAI'04)*, Nov. 2004.

[26] WEKA: http://www.cs.waikato.ac.nz/ml/weka/

[27] I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementation*, second ed., Morgan Kaufmann, 2005.

[28] C. C. Williams and J. K. Hollingsworth, Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques, *IEEE Trans. Software Engineering,* vol. 31, pp. 466-480, 2005.

[29] X. Wu, *Knowledge Acquisition from Databases*, Ablex Publishing, 1995.

[30] X. Zhu and X. Wu, Class Noise vs. Attribute Noise: A Quantitative Study of Their Impacts, *Artificial Intelligence Review* 22: 177–210, Kluwer Academic, 2004.

[31] H. Zhang and X. Zhang, Comments on "Data Mining Static Code Attributes to Learn Defect Predictors", *IEEE Trans. on Software Engineering*, 33(9), 635-636, 2007.

[32] H. Zhang, X. Zhang and M. Gu, Predicting Defective Software Components from Code Complexity Measures, *Proc. of 13th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, Dec 2007, Australia.

[33] H. Zhang, An Investigation of the Relationships between Lines of Code and Defects, *Proc. ICSM'09*, Edmonton, Canada, September 2009.

[34] T. Zimmermann, R. Premraj and A. Zeller, 2007. Predicting Defects for Eclipse, *Proc. PROMISE'07*, Minneapolis, USA.

[35] T. Zimmermann and N. Nagappan, Predicting Defects using Network Analysis on Dependency Graphs, *Proc. ICSE'08*, Leipzig, Germany, May 2008.