A General Software Defect-Proneness Prediction Framework

Qinbao Song, Zihan Jia, Martin Shepperd, Shi Ying and Jin Liu

Abstract—BACKGROUND – predicting defect-prone software components is an economically important activity and so has received a good deal of attention. However, making sense of the many, and sometimes seemingly inconsistent, results is difficult.
OBJECTIVE – we propose and evaluate a general framework for software defect prediction that supports (i) unbiased and (ii) comprehensive comparison between competing prediction systems.
METHOD – the framework comprises (i) scheme evaluation and (ii) defect prediction components. The scheme evaluation analyzes the prediction performance of competing learning schemes for given historical data sets. The defect predictor builds models according to the evaluated learning scheme and predicts software defects with new data according to the constructed model. In order to demonstrate the performance of the proposed framework, we use both simulation and publicly available software defect data sets.
RESULTS – the results show that we should choose different learning schemes for different data sets (i.e. no scheme dominates), that small details in conducting how evaluations are conducted can completely reverse findings and lastly that our proposed framework is more effective, and less prone to bias than previous approaches.
CONCLUSIONS – failure to properly or fully evaluate a learning scheme can be misleading, however, these problems may be overcome by our proposed framework.

Key Words—Software defect prediction, software defect-proneness prediction, machine learning, scheme evaluation.

1 Introduction

Software defect prediction has been an important research topic in the software engineering field for more than 30 years. Current defect prediction work focuses on (i) estimating the number of defects remaining in software systems, (ii) discovering defect associations, and (iii) classifying the defect-proneness of software components, typically into two classes defect-prone and not defect-prone. This paper is concerned with the third approach.

The first type of work employs statistical approaches [1], [2], [3], capture-recapture (CR) models [4], [5], [6], [7], and detection profile methods (DPM) [8] to estimate the number of defects remaining in software systems with inspection data and process quality data. The prediction result can be used as an important measure for the software developer [9], and can be used to control the software process (i.e. decide whether to schedule further inspections or pass the software artifacts to the next development step [10]) and gauge the *likely* delivered quality of a software system [11].

The second type of work borrows association rule mining algorithms from the data mining community to reveal software defect associations [12], which can be used for three purposes. First to find as many related defects as possible to the detected defect(s) and consequently make more effective corrections to the software. This may be useful as it permits more directed testing and more effective use of limited testing resources. Second, to help evaluate reviewers' results during an inspection. Thus a recommendation might be that his/her work should be reinspected for completeness. Third, to assist managers in improving the software process through analysis of the reasons why some defects frequently occur together. If the analysis leads to the identification of a process problem, managers can devise corrective action.

The third type of work classifies software components as defect-prone and non-defect-prone by means of metric-based classification [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]. Being able to predict which components are more likely to be defect-prone supports better targeted testing resources and therefore improved efficiency.

Unfortunately, classification remains a largely unsolved problem. In order to address this researchers have been using increasingly sophisticated techniques drawn from machine learning. This sophistication has led to challenges in how such techniques are configured and how they should be validated. Incomplete or inappropriate validation can result in unintentionally misleading results and over-optimism on the part of the researchers. For this reason we propose a new and more general framework within which to conduct such validations. To reiterate a comment made in an earlier paper by one of the authors [MS] and also quoted by Lessmann et al.

- *Q. Song and Z. Jia are with the Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, 710049 China.*
  E-mail: qbsong@mail.xjtu.edu.cn, jiazh.eden@stu.xjtu.edu.cn.
- *M. Shepperd is with the School of Information Science, Computing, and Mathematics, Brunel University, Uxbridge, UB8 3PH UK.*
  E-mail: martin.shepperd@brunel.ac.uk.
- *S. Ying and J. Liu are with the State Key Laboratory of Software Engineering, Wuhan University, Wuhan, 430072 China.*
  E-mail: yingshi@whu.edu.cn, mailjinliu@yahoo.com.

Digital Object Indentifier 10.1109/TSE.2010.90 0098-5589/10/$26.00 © 2010 IEEE

[24] "we need to develop more reliable research procedures before we can have confidence in the conclusion of comparative studies of software prediction models" [25]. Thus we stress that the aim of this paper is to consider how we evaluate different *processes* for finding classification models, not any particular model itself. We consider it most unlikely any useful, universal model exists.

Much of this research activity has followed the path of using software metrics extracted from the code as candidate factors to reveal whether a software component is defect-prone or not. To accomplish this a variety of machine learning algorithms have been used to inductively find patterns or rules within the data to classify software components as either defect-prone or not. Examples include [13], [26], [27], [28], [20], [23], and [24]. In addition Wagner [29] and Runeson et al. [30] provide useful overviews in the form of systematic literature reviews.

In order to motivate the need for more systematic and unbiased methods for comparing the performance of machine learning based defect prediction we focus on a recent paper published in this journal by Menzies, Greenwald and Frank [23]. For brevity we will refer to this as the MGF paper. We choose MGF for three reasons. First, because it has been widely cited[1] and is therefore influential. Second, because the approach might be regarded as state of the art for this kind of research. Third, because the MGF analysis is based upon datasets in the public domain thus we are able to replicate the work. We should stress we are *not* singling this work out for being particularly outrageous. Instead we wish to respond to their challenge "that numerous researchers repeat our experiments and discover learning methods that are superior to the one proposed here" (MGF).

In the study, publicly available datasets from different organizations are used. This allows us to explore the impact of data from different sources on different processes for finding appropriate classification models apart from evaluating these processes in a fair and reasonable way. Additionally, 12 learning schemes[2] resulted from two data preprocessors, two feature selectors, and three classification algorithms are designed to assess the effects of different elements of a learning scheme on defect prediction. Although *balance* is a uncommon measure in classification, the results of MGF were reported with it, thus it is still used whilst a general measure AUC of predictive power is employed in the paper as well.

This paper makes the following contributions: (i) a new and more general software defect-proneness prediction framework within which appropriate validations can be conducted is proposed; (ii) the impacts of different elements of a learning scheme on the evaluation and prediction are explored, and concluded that a learning scheme should be evaluated as holistically and no learning scheme dominates, consequently the evaluation and decision process is important; and (iii) the potential bias and misleading results of the MGF framework is explained and confirmed, and demonstrated that the performance of the MGF framework is varying greatly with data from different organizations.

The remainder of the paper is organized as follows. Section 2 provides some further background on the current state of the art for learning software defect prediction systems with particular reference to MGF. Section 3 describes our framework in detail and analyzes differences between our approach and that of MGF. Section 4 is devoted to the extensive experiments to compare our framework and that of MGF and to evaluate the performance of the proposed framework. Conclusions and consideration of the significance of this work are given in the final section.

2 Related work

MGF [23] published a study in this journal in 2007 in which they compared the performance of two machine learning techniques (Rule Induction and Naïve Bayes) to predict software components containing defects. To do this they use the NASA MDP repository which at the time of their research contained 10 separate data sets.

Traditionally many researchers have explored issues like the relative merits of McCabe's cyclomatic complexity, Halstead's software science measures and lines of code counts for building defect predictors. However, MGF claim that "such debates are irrelevant since *how* the attributes are used to build predictors is much more important than *which* particular attributes are used" and "the choice of learning method is far more important than which subset of the available data is used for learning". Their analysis found that a Naïve Bayes classifier, after *log-filtering* and attribute selection based on *InfoGain* had a mean probability of detection of 71% and mean false alarms rates of 25%. This significantly outperformed the rule induction methods of J48 and OneR (due to Quinlan [31]).

We argue that although *how* is more important than *which*[3], the choice of which attribute subset is used for learning is not only circumscribed by the attribute subset itself and available data, but also by attribute selectors, learning algorithms and data preprocessors. It is well known that there is an intrinsic relationship between a learning method and an attribute selection method. For example, Hall and Holmes [32] concluded that the forward selection search was well suited to Naïve Bayes but the backward elimination search is more suitable for C4.5. Cardie [33] found using a decision tree to select attributes helped the nearest neighbor algorithm to reduce its prediction error. Kubat et al. [34] used a decision tree

[1] Google scholar (accessed February 6, 2010) indicates an impressive 132 citations to MGF [23] within the space of three years.

[2] Please see Section 2 for the details of a learning scheme.

[3] That is, which attribute subset is more useful for defect prediction not only depends on the attribute subset itself but also on the specific data set.

filtering attributes for use with a Naïve Bayesian classifier and obtained a similar result. However, Kibler and Aha [35] reported more mixed results on two medical classification tasks. Therefore, before building prediction models, we should choose the combination of all three of learning algorithm, data pre-processing and attribute selection method, not merely one or two of them.

Lessmann et al. [24] have also conducted a follow-up to MGF on defect predictions, providing additional results as well as suggestions for a methodological framework. However, they did not perform attribute selection when building prediction models. Thus our work has wider application.

We also argue that MGF's attribute selection approach is problematic and yielded a bias in the evaluation results, despite the use of a M×N-*way cross-evaluation* method. One reason is that they ranked attributes on the entire data set including both the training and test data, though the class labels of the test data should have been unknown to the predictor. That is, they violated the intention of the holdout strategy. The potential result is that they overestimate the performance of their learning model and thereby report a potentially misleading result. Moreover, after ranking attributes, they evaluated each individual attribute separately and chose those *n* features with the highest scores. Unfortunately, this strategy cannot consider features with complementary information, and does not account for attribute dependence. It is also incapable of removing redundant features because redundant features are likely to have similar rankings. As long as features are deemed relevant to the class, they will all be selected even though many of them are highly correlated to each other.

These seemingly minor issues motivate the development of our general-purpose defect prediction framework described in this paper. However, we will show the large impact they can have and how researchers may be completely misled. Our proposed framework consists of two parts: scheme evaluation and defect prediction. The scheme evaluation focuses on evaluating the performance of a learning scheme, whilst the defect prediction focuses on building a final predictor using historical data according to the learning scheme and after which the predictor is used to predict the defect-prone components of a new (or unseen) software system.

A learning scheme comprises:

1) a data preprocessor,
2) an attribute selector,
3) a learning algorithm.

So to summarize, the main difference between our framework and that of MGF lies in: (i) we choose the entire learning scheme, not just one out of the learning algorithm, attribute selector or data pre-processor; (ii) we use the appropriate data to evaluate the performance of a scheme. That is, we build a predictive model according to a scheme with only 'historical' data and validate the model on the independent 'new' data. We go on to demonstrate why this has very practical implications.

3 Proposed Software Defect Prediction Framework

3.1 Overview of the framework

Generally, before building defect prediction model(s) and using them for prediction purposes, we first need to decide which learning scheme should be used to construct the model. Thus the predictive performance of the learning scheme(s) should be determined, especially for future data. However, this step is often neglected and so the resultant prediction model may not be trustworthy. Consequently we propose a new software defect prediction framework that provides guidance to address these potential shortcomings. The framework consists of two components: (i) scheme evaluation and (ii) defect prediction. Fig. 1 contains the details.

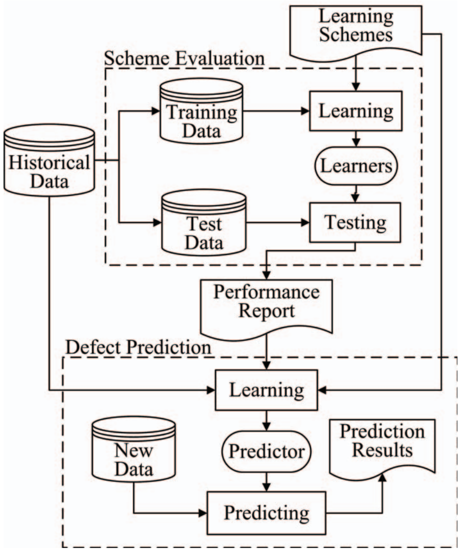


Fig. 1. Proposed software defect prediction framework

At the scheme evaluation stage, the performances of the different learning schemes are evaluated with historical data to determine whether a certain learning scheme performs sufficiently well for prediction purposes or to select the best from a set of competing schemes.

From Fig. 1 we can see that the historical data are divided into two parts: a training set for building learners with the given learning schemes, and a test set for evaluating the performances of the learners. It is very important that the test data are not used in *any way* to build the learners. This is a necessary condition to assess the generalization ability of a learner that is built according to a learning scheme, and further to determine whether or not to apply the learning scheme, or select one best scheme from the given schemes.

At the defect prediction stage, according to the performance report of the first stage, a learning scheme is selected and used to build a prediction model and predict software defect. From Fig. 1 we observe that all the historical data are used to build the predictor here. This is very different from the first stage; it is very useful for improving the generalization ability of the predictor. After the predictor is built, it can be used to predict the defect-proneness of new software components.

MGF proposed a baseline experiment and reported the performance of the Naïve Bayes data miner with *log-filtering* as well as attribute selection, which performed the scheme evaluation but with inappropriate data. This is because they used both the training (which can be viewed as historical data) and test (which can be viewed as new data) data to rank attributes, while the labels of the new data are unavailable when choosing attributes in practice.

3.2 Scheme evaluation

The scheme evaluation is a fundamental part of the software defect prediction framework. At this stage, different learning schemes are evaluated by building and evaluating learners with them. Fig. 2 contains the details.

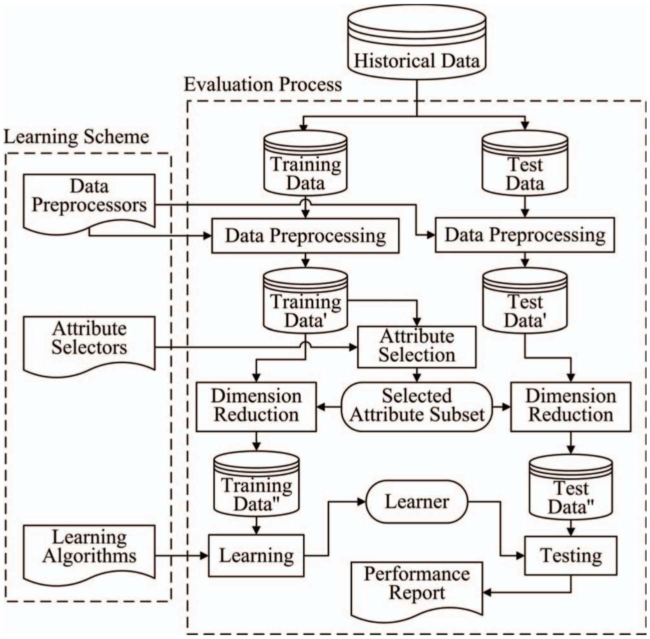


Fig. 2. Scheme evaluation of the proposed framework.

The first problem of scheme evaluation is how to divide historical data into training and test data. As mentioned above, the test data should be independent of the learner construction. This is a necessary precondition to evaluate the performance of a learner for new data. Cross-validation is usually used to estimate how accurately a predictive model will perform in practice. One round of cross-validation involves partitioning a data set into complementary subsets, performing the analysis on one subset, and validating the analysis on the other subset. To reduce variability, multiple rounds of cross-validation are performed using different partitions, and the validation results are averaged over the rounds.

In our framework, a M×N-*way cross-validation* is used for estimating the performance of each predictive model, that is, each data set is first divided into N bins, and after that a predictor is learned on (N-1) bins, and then tested on the remaining bin. This is repeated for the N folds so that each bin is used for training and testing while minimizing the sampling bias. To overcome any ordering effect and to achieve reliable statistics, each holdout experiment is also repeated M times and in each repetition the data sets are randomized. So overall, M×N models are built in all during the period of evaluation, thus M×N results are obtained on each data set about the performance of the each learning scheme.

After the training-test splitting is done each round, both the training data and learning scheme(s) are used to build a learner. A learning scheme consists of a data preprocessing method, an attribute selection method, and a learning algorithm. The detailed learner construction procedure is as follows:

1) Data preprocessing
   This is an important part of building a practical learner. In this step, the training data are preprocessed, such as removing outliers, handling missing values, discretizing or transforming numeric attributes. In our experiment, we use a *log-filtering* preprocessor which replaces all numerics n with their logarithms *ln(n)* such as used in MGF.
2) Attribute selection
   The data sets may not have originally been intended for defect prediction, thus even if all the attributes are useful for its original task, not all may be helpful for defect prediction. Therefore attribute selection has to be performed on the training data. Attribute selection methods can be categorized as either filters or wrappers [36]. It should be noted that both 'filter' and 'wrapper' methods only operate on the training data. A 'filter' uses general characteristics of the data to evaluate attributes and operates independently of any learning algorithm. In contrast, a 'wrapper' method, exists as a wrapper around the learning algorithm searching for a good subset using the learning algorithm itself as part of the function evaluating attribute subsets. Wrappers generally give better results than filters but are more computationally intensive. In our proposed framework, the 'wrapper' attribute selection method is employed. To make most use of the data, we use a M×N-*way cross-validation* to evaluate the performance of different attribute subsets.
3) Learner construction
   Once attribute selection is finished, the preprocessed training data are reduced to the *best* attribute subset. Then the reduced training data and the learning algorithm are used to build the learner. Before the learner is tested, the original test data are preprocessed in the same way and the dimensionality is reduced to the same *best* subset of attributes. After comparing the predicted value and the actual value of the test data, the performance of one *pass*

of validation is obtained. As mentioned previously, the final 'evaluation' performance can be obtained as the mean and variance values across the M×N *passes* of such validation.

The detailed scheme evaluation process is described with pseudocode in the following Procedure *Evaluation* which consists of Function *Learning* and Function *AttrSelect*. The Function *Learning* is used to build a learner with a given learning scheme, and the Function *AttrSelect* performs attribute selection with a learning algorithm.

3.3 Defect prediction

The defect prediction part of our framework is straightforward, it consists of predictor construction and defect prediction.

During the period of the predictor construction,

1) A learning scheme is chosen according to the Performance Report.
2) A predictor is built with the selected learning scheme and the *whole* historical data. While evaluating a learning scheme, a learner is built with the training data and tested on the test data. Its final performance is the mean over all rounds. This reveals that the evaluation indeed covers all the data. However, a single round of cross-validation uses only one part of the data. Therefore, as we use all the historical data to build the predictor, it is expected that the constructed predictor has stronger generalization ability.
3) After the predictor is built, new data are preprocessed in same way as historical data, then the constructed predictor can be used to predict software defect with preprocessed new data.

The detailed defect prediction process is described with pseudocode in the following Procedure *Prediction*.

3.4 Difference between our proposed framework and MGF

Although both MGF's and our study have involved a M×N-*way cross-validation* there is, however, a significant difference. In their study, for each data set, the attributes were ranked by *InfoGain* which was calculated on the whole data set, then the M×N-*way validation* was wrapped inside scripts that explored different subset of attributes in the order suggested by the *InfoGain*. In our study, there is an M×N-*way cross-validation* for performance estimation of the learner with attribute selection, which is out of the attribute selection procedure. We only performed attribute selection on the training data. When a 'wrapper' selection method is performed, another cross-validation can be performed to evaluate the performance of different attribute subsets. This should be performed only on the training data.

To recap, the essential problem in MGF's study is that the test data were used for attribute selection, which actually violated the intention of holdout strategy. In their study, the M×N-*way cross-validation* actually implemented a holdout strategy to just select the 'best' subset among the subsets recommended by *InfoGain* for each data set. However, as the "test data" is unknown at that period of time, so the result obtained in that way potentially overfits the current data set itself and cannot be used to assess the future performance of the learner built with such 'best' subset.

Our framework focuses on the attribute selection method itself instead of certain 'best' subset, as different training data may produce different best subsets. We treat the attribute selection method as a part of the learning scheme. The 'inner' cross-validation is performed on the training data, which actually selects the 'best' attribute set on the training data with the basic learning algorithm. After that, the 'outer' cross-validation assesses how well the learner built with such 'best' attributes performs on the test data, which is really new to the learner. Thus our framework can properly assess the future performance of the learning scheme as a whole.

4 Empirical Study

4.1 Data sets

We used the data taken from the public NASA MDP repository [37], which was also used by MGF and many others e.g. [24], [38], [39], and [22]. What's more, the AR data from the PROMISE repository[4] was also used. Thus there are 17 data sets in total, 13 from NASA and the remaining 4 from the PROMISE repository.

Table 1 provides some basic summary information. Each data set comprises a number of software modules (cases), each containing the corresponding number of defects and various software static code attributes. After preprocessing, modules that contain one or more defects were labeled as defective. Besides LOC counts, the data sets include Halstead attributes as well as McCabe complexity measures[5]. A more detailed description of code attributes or the origin of the MDP data sets can be obtained from [23].

4.2 Performance measures

The *receiver operating characteristic* (ROC) curve is often used to evaluate the performance of binary predictors. A typical ROC curve is shown in Fig. 3. The y-axis shows probability of detection (*pd*) and the x-axis shows probability of false alarms (*pf*).

Formal definitions for *pd* and *pf* are given in Equations 1 and 2 respectively. Obviously higher *pd*s and lower *pf*s are desired. The point ($pf = 0$, $pd = 1$) is the ideal

[4] http://promise.site.uottowa.ca/SERepository

[5] Whilst there is some disquiet concerning the value of such code metrics recall that the purpose of this paper is to examine frameworks for learning defect classifiers and not to find the 'best' classifier *per se*. Moreover, since attribute selection is part of the framework we can reasonably expect irrelevant or redundant attributes to be eliminated from any final classifier.

```
Procedure Evaluation(historicalData, scheme)
input : historicalData - the historical data;
        scheme - the learning scheme.
output: AvgResult - the mean performance over the M×N-way cross-validation.

M = 10 ; /*number of repetitions                                          */
N = 10 ; /*number of folds                                                */
repeat
    D = Randomize (historicalData); /*randomize the order of instances    */
    Generate N bins from D ;
    for $i = 1$ to N do
        test = bin[$i$];
        train = D − test ;
        [learner, bestAttrs ] = Learning (train, scheme);
        test′ = select bestAttrs from test ;
        Result = TestClassifier (test′, learner);
        /*Compute the performance measures of the learner on data test′  */
    end
until M times ;
AvgResult = $\frac{1}{\mathbf{M}\times\mathbf{N}}\sum$Result ;
```

```
Function Learning(data, scheme)
input : data - the data on which the learner is built;
        scheme - the learning scheme.
output: learner - the final learner built on data with scheme;
        bestAttrs - the best attribute subset selected by the attribute selector of scheme

m = 10 ; /*number of repetitions for attribute selection                  */
n = 10 ; /*number of folds for attribute selection                        */
d = Preprocessing (data, scheme.preprocessor);
bestAttrs = AttrSelect (d, scheme.algorithm, scheme.attrSelector, m, n);
d′ = select bestAttrs from d ;
learner = BuildClassifier (d′, scheme.algorithm);
/*build a classifier on d′ with the learning algorithm of scheme          */
```

```
Procedure Prediction(historicalData, newData, scheme)
input : historicalData - the historical data; newData - the new data;
        scheme - the learning scheme.
output: Result - the predicted result for the newData

[predictor, bestAttrs ] = Learning (historicalData, scheme);
d = select bestAttrs from newData;
Result = Predict (d, predictor);
/*predict the class label of d with predictor                             */
```

position where we recognize all defective modules and never make mistakes.

$$pd = tpr = \frac{TP}{TP + FN} \tag{1}$$

$$]pf = fpr = \frac{FP}{FP + TN} \tag{2}$$

$$balance = 1 - \frac{\sqrt{(1 - pd)^2 + (0 - pf)^2}}{\sqrt{2}} \tag{3}$$

MGF introduced a performance measure called *balance* which is used to choose the optimal (*pd*, *pf*) pairs. The definition is shown in Equation 3, from which we can see that it is equivalent to the normalized Euclidean distance from the desired point (0,1) to (*pf*, *pd*) in a ROC curve [40]. In order to more directly and fairly compare our results with that of MGF, *balance* is used as a performance measure.

Zhang and Zhang [41] argue that using (*pd*, *pf*) performance measures in the classification of imbalanced data is not practical due to low precisions. By contrast,

TABLE 1
Code Attributes within NASA MDP and AR Data Sets

	NASA MDP Dataset													*AR Dataset*			
	cm1	jm1	kc1	kc3	kc4	mw1	mc1	mc2	pc1	pc2	pc3	pc4	pc5	ar1	ar3	ar4	ar6
LOC_total	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
LOC_blank	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
LOC_code_and_comment	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
LOC_comments	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
LOC_executable	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Number_of_lines	x			x	x	x	x	x	x	x	x	x					
num_operators	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
num_operands	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
num_unique_operators	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
num_unique_operands	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Halstead_Length	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Halstead_Volume	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Halstead_Level	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Halstead_Difficulty	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Halstead_Content	x	x	x	x	x	x	x	x	x	x	x	x	x				
Halstead_Effort	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Halstead_Error_Est	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Halstead_Prog_Time	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Halstead_vocabulary														x	x	x	x
cyclomatic_complexity	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
cyclomatic_density							x	x					x	x	x	x	x
design_complexity	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
essntial_complexity	x	x	x	x	x	x	x	x	x	x	x	x	x				
essential_density							x	x					x				
branch_count	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
call_pairs	x			x	x	x	x	x	x	x	x	x	x	x	x	x	x
condition_count	x			x	x	x	x	x	x	x	x	x	x	x	x	x	x
decision_count	x			x	x	x	x	x	x	x	x	x	x	x	x	x	x
decision_density	x			x	x	x		x	x	x	x	x		x	x	x	x
design_density	x			x	x	x	x	x	x	x	x	x	x	x	x	x	x
edge_count	x			x	x	x	x	x	x	x	x	x	x				
global_data_complexity	x			x	x	x	x	x	x	x	x	x	x				
global_data_density	x			x	x	x	x	x	x	x	x	x	x				
maintenance_severity	x			x	x	x	x	x	x	x	x	x	x				
modified_condition_count	x			x	x	x	x	x	x	x	x	x	x				
multiple_condition_count	x			x	x	x	x	x	x	x	x	x	x	x	x	x	x
node_count	x			x	x	x	x	x	x	x	x	x	x				
normalized_cylomatic_compl	x			x	x	x	x	x	x	x	x	x	x	x	x	x	x
parameter_count	x			x	x	x	x	x	x	x	x	x	x				
formal_parameters														x	x	x	x
pathological_complexity	x			x	x	x	x	x	x	x	x	x	x				
percent_comment	x			x	x	x	x	x	x	x	x	x	x				
number of attributes	38	21	21	38	38	38	39	40	38	38	38	38	39	29	29	29	29
number of modules	505	10878	2107	458	125	403	9466	161	1107	5589	1563	1458	17186	121	63	107	101
number of fp modules	48	2102	325	43	61	31	68	52	76	23	160	178	516	9	8	20	15
percentage of fp modules	9.50	19.32	15.42	9.39	48.80	7.69	0.72	32.30	6.87	0.41	10.24	12.21	3.00	7.44	12.70	18.69	14.85

MGF argue that precision has an unstable nature and can be misleading to determine the better predictor. We also think that predictors with high *pd* have practical usage even when their *pf* is also high. Nevertheless such a predictor could still be helpful for software testing, especially in mission critical and safety critical systems, where the cost of many false positives (wrongly identifying a software component as fault-prone) is far less than that of false negatives [42], [43], [44].

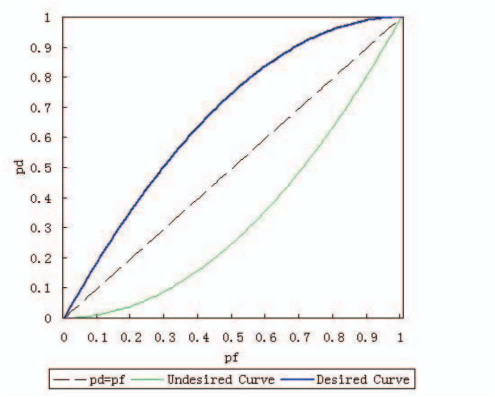


Fig. 3. The ROC curves.

However, we would like to note that *balance* should be used carefully for determining the best among a set of predictors. Since it is a distance measure, predictors with different (*pf*, *pd*) values can have the same *balance* value. Nevertheless, this doesn't necessarily show that all predictors with the same *balance* value have the same practical usage. Usually, domain specific requirement may lead us to choose a predictor with a high *pd* rank although it may also have a high *pf* rank. For an extensive discussion of different approaches to model performance indicators see [40].

Moreover, *balance* is based on the assumption that there is no difference between the cost of false positives (FP) and false negatives (FN). That means the module x is seen as defect-prone when $p(y = Defective|x) > p(y = Nondefective|x)$ provided by the predictor. However, when different costs of FP and FN are considered, the condition for judging module x as such becomes

$$\frac{p(y = Defective|x)}{p(y = Nondefective|x)} > \frac{C_{FP}}{C_{FN}}, \tag{4}$$

where C_{FP} denotes the cost of an FP error (classifying a nondefective module incorrectly as defective), and C_{FN} denotes the cost of an FN error (misclassifying a defective module). As is known that $p(y = Defective|x) + p(y = Nondefective|x) = 1$, the condition is equivalent to

$$p(y = Defective|x) > \frac{C_{FP}}{C_{FN} + C_{FP}}. \tag{5}$$

Thus it can seen that different results can be obtained for the same predictor when different discrimination thresholds are used. Each point on the ROC curve is mapped to a discrimination threshold. The point (0,0) means the predictor treats all modules as nondefective, namely the corresponding threshold is 1. And the point (1,1) means the predictor treats all as defective, namely the corresponding threshold is 0. Thus the ROC curve characterizes the performance of a binary predictor among varied threshold. As shown in Fig. 3, "desired curves" bend up towards this ideal point (0,1) and "undesired curves" bend away from the ideal point.

The AUC (Area Under ROC Curve) is often calculated to compare different ROC curves. Higher AUC values indicate the classifier is on average more to the upper left region of the graph. AUC represents the most informative and commonly used, thus it is used as another performance measure in this paper.

Next, in order to more clearly compare the evaluation performance of the two frameworks, we use *diff* which is defined as follows.

$$\mathit{diff} = \frac{\mathit{EvaPerf} - \mathit{PredPerf}}{\mathit{PredPerf}} \times 100\% \tag{6}$$

where *EvaPerf* represents the mean evaluation performance and *PredPerf* denotes the mean prediction performance. The performance measure can be either *balance* or AUC.

From the definition we can know that (i) a positive *diff* means that the mean evaluation performance is higher than the mean prediction performance, so the evaluation is optimistic; (ii) a negative *diff* means a lower mean evaluation performance than corresponding mean prediction performance, thus the evaluation is conservative. No matter whether the *diff* is positive or negative, it is clear that the smaller the absolute value of *diff* is, the more accurate the evaluation is. And this is our main goal, to have a framework that minimizes this gap.

4.3 Experiment Design

Two experiments are designed in the experiment. One is to compare our framework with that of MGF, the second is intended to demonstrate our framework in practice and explore whether we should choose a particular learning scheme or not.

4.3.1 Framework comparison

This experiment was used to compare our framework with that of MGF who reported that a Naïve Bayes data miner with a *log-filtering* preprocessor achieved a mean (*pd*, *pf*)$= (71, 25)$. As discussed in Section 2, their study might potentially overfit the *historical* data and the results reported could be misleading. This suggests that the actual performance of the predictor might not be so good when used to predict using *new* data.

In our experiment, we simulated the whole process of defect prediction to explore whether MGF's evaluation result is misleading or not. The experimental process is described as follows:

1) We divided each data set into two parts: one is used as *historical* data and the other is viewed as *new* data. To make most use of the data, we performed 10-*pass* simulation. In each pass, we took 90% of the data as *historical* data, the remaining 10% as *new* data.
2) We replicated MGF's work with the *historical* data. Firstly, all the *historical* data was preprocessed in the same way by a *log-filtering* preprocessor. Then an *iterative* attribute subset selection as used in MGF's study was performed. In the subset selection method, the i = 1, 2, ..., N-th top-ranked attribute(s) were evaluated step by step. Each subset was evaluated by a 10×10-*way cross-validation* with the Naïve Bayes algorithm, the averaged *balance* after 100 holdout experiments was used to estimate the performance. The process of attribute subset selection was terminated when the first $i{+}1$ attributes performed no better than the first i. So the first i top-ranked attributes were selected as the best subset, with the averaged *balance* as the evaluation performance.
   The *historical* data was processed by the *log-filtering* method and reduced by the selected best attribute subset and the resultant data were used to build a Naïve Bayes predictor. Then the predictor was used to predict defect with the *new* data that was processed by same way as that of the *historical* data.
3) We also simulated the whole defect prediction process presented in our framework.
   In order to be comparable with MGF, we restricted our learning scheme to the same preprocessing method, attribute selection method and the same learning algorithm. A 10×10-*way cross-validation* was used to evaluate the learning scheme. The learning scheme was wrapped in each validation of the 10×10-*way cross-validation*, which is different from MGF's study. Specifically, as described in the scheme evaluation procedure, we applied the learning scheme only to the training data, after which the final Naïve Bayes learner was built and the test data were used to evaluate the performance of the learner. One hundred such holdout experiments were performed for each pass of the

evaluation and the mean of 100 *balance* measures was reported as the evaluation performance.

The *historical* data was processed according to the learning scheme, and a Naïve Bayes predictor was built with the processed data. Then the predictor was used to predict defect with the *new* data that was processed by the same way as that of the *historical* data.

4.3.2 Defect prediction with different learning schemes

This experiment is intended to demonstrate our framework and to illustrate that different elements of a learning scheme have different impacts on the predictions and to confirm that we should choose the combination of a data preprocessor, an attribute selector and a learning algorithm, instead of any one of them separately.

For this purpose, twelve different learning schemes were designed according to the following data preprocessors, attribute selectors and learning algorithms.

1) Two data preprocessors
    - a) None: data unchanged;
    - b) Log: all the numeric values are replaced by their logarithmic values, as used by MGF.
2) Two attribute selectors
    The standard wrapper method is employed to choose attributes. This means the performances of the learning algorithms are used to evaluate the selected attributes. Two different search strategies (based on greedy algorithms) are used as follows.
    - a) Forward selection (FS): starts from an empty set and evaluates each attribute individually to find the best single attribute. It then tries each of the remaining attributes in conjunction with the best to find the best pair of attributes. In the next iteration each of the remaining attributes are tried in conjunction with the best pair to find the best group of three attributes. This process continues until no single attribute addition improves the evaluation of the subset.
    - b) Backward elimination (BE): starts with the whole set of attributes, and eliminates one attribute in each iteration until no single attribute elimination improves the evaluation of the subset.
3) Three learning algorithms
    Naïve Bayes (NB), J48[6] and OneR.
4) Twelve learning schemes
    The combination of two data preprocessors, two attribute selectors and three learning algorithms, yields total twelve different learning schemes:
    a) NB+Log+FS; b) J48+Log+FS; c) OneR+Log+FS; d) NB+Log+BE; e) J48+Log+BE; f) OneR+Log+BE; g) NB+None+FS; h) J48+None+FS; i) OneR+None+FS; j) NB+None+BE; k) J48+None+BE; and l) OneR+None+BE.

TABLE 2
Framework comparison for all the data sets

Data set	MGF Framework		Our Framework	
	Eval	Pred	Eval	Pred
CM1	72.7	69.5	68.3	69.5
KC3	74.1	69.7	69.4	**70.8**
KC4	71.9	68.1	66.1	**69.1**
MW1	70.5	66.5	64.8	66.1
PC1	64.6	62.7	66.0	**66.8**
PC2	81.8	76.2	78.1	**79.7**
PC3	71.4	71.0	70.3	**71.1**
PC4	82.6	82.2	82.0	82.1
JM1	44.0	43.9	58.5	**58.5**
KC1	70.5	70.7	70.2	70.7
MC1	82.8	79.7	80.1	79.3
MC2	61.0	59.6	57.7	**61.4**
PC5	89.7	88.8	90.4	**90.4**
AR1	53.8	42.8	39.8	41.1
AR3	80.7	66.1	60.0	66.1
AR4	71.7	66.4	64.1	**68.3**
AR6	53.1	47.0	51.9	**49.2**

In this experiment, each data set was still divided into two parts: one is used as *historical* data and the other viewed as *new* data. To make the most use of the data, we performed 10-*pass* simulation. For each pass, we took 90% of the data as *historical* data, and the remainder as *new* data. We performed the whole process twice, with *balance* and AUC respectively.

4.4 Experimental results and analysis

4.4.1 Framework comparison

The framework comparison results are summarized in Table 2 [7] which shows the results in terms of *balance*. From it we find that our framework outperformed that of MGF for 10 out of 17 data sets with a further 3 ties (each 'winner' is denoted in bold).

The mean prediction *balance* of the MGF framework over the 17 data sets is 66.5%, and the mean prediction *balance* of the proposed framework is 68.2%, with an improvement of 2.6%. We confirm this formally by the Wilcoxon signed-rank test of medians (a non-parametric alternative of the t-test) which yields $p = 0.0040$ for a 1-tailed hypothesis that the new framework is superior to the MGF framework.

However, the point we wish to make is more subtle than merely which technique is 'best', we want to show that it is how one answers this question that really matters. As we have previously indicated the MGF approach contains a bias to over-optimism, thus if we restrict our comparison to the 'Eval' columns of Table 2 we would see a different (and misleading) picture. First, using the *balance* performance indicator, we find that MGF outperforms our approach for 14 out of 17

[6] J48 is a JAVA implementation of Quinlan's C4.5 (version 8) algorithm [31].

[7] The 'Eval' columns show the evaluation performance on each data set, while the 'Pred' columns show the prediction performance.

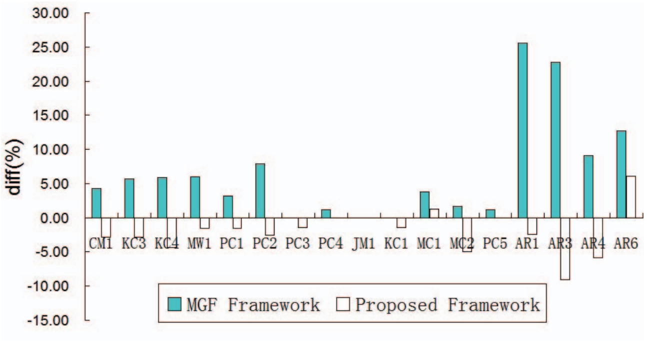


Fig. 4. The *balance diff*s of the two frameworks.

data sets. In this case the same Wilcoxon signed-rank test would have been rejected at $p = 1.0$ and indeed the opposite hypothesis, that the MGF framework is superior to our framework at $p < 0.0001$. Thus we see what a dramatic impact a seemingly small difference in a validation procedure can have.

Finally we note the considerable variation in the performances across the data sets. This yet again suggests that a simple search for the 'best' predictor is likely to be a pointless endeavour.

Fig. 4 shows the *balance diff*s of the two frameworks on the 17 data sets. From Fig. 4 we observe that

1) For MGF framework, the *balance diff* values are always positive except for the KC1 data. This means the evaluation performance of MGF framework is always higher than the prediction performance. This reveals they overestimated the performance and the result they reported in [23] may be misleading.
2) For the proposed framework, the *balance diff* values are always negative except for the JM1 and PC5 data are zero, and MC1 and AR6 data are positive. This reveals that our evaluation is a little conservative. The potential reason is that the training data used for learner building in the evaluation is just 90% of the '*historical*' data, while the final predictor is built on the whole '*historical*' data. That is, the final predictor is built with more sufficiency data than the learner built in the evaluation, hence has a higher *balance* on average.
3) The largest absolute value of *balance diff* in MGF framework is 25.7% on AR1 data, on which the corresponding absolute value of *balance diff* in the proposed framework is just 3.16%. Moreover, the largest absolute value of *balance diff* in the proposed framework is 9.23% on AR3 data, on which the corresponding absolute value of *balance diff* in MGF framework is over 22%. Also, the mean absolute *balance diff* value of the proposed framework over the 17 data sets is 2.72%, which is much smaller than MGF's 6.52%. Finally a Wilcoxon signed-rank test of medians yields $p = 0.0028$ for a 1-tailed hypothesis that the absolute *balance diff* of the new framework is significantly less than that of the MGF framework.

On the other hand, comparing the *balance diff* on the datasets from the two different organizations, we observe that for the MGF framework, the mean absolute value of the *balance diff* on the NASA data is 3.13%, which is much smaller than the 17.54% on the AR data, the difference is 14.41%; while in our framework, the mean absolute value of *balance diff* on the NASA data is 1.89%, which is smaller than the 5.88% on the AR data, the difference is only 3.99% which is much smaller than that of the MGF framework. This reveals that the predictions of Menzies et al. on the AR data are much more biased than that on the NASA data and the performance of the MGF framework is varying greatly with data from different organizations.

To summarize, the above experiment results show that the performance evaluation of MGF is too optimistic, this means the real prediction performance is unlikely to be so good as the evaluation performance. The proposed framework is a little conservative. However, the evaluation of the latter is closer to the real performance than the former. On the other hand, the mean prediction performance of the proposed framework is higher than that of MGF. This indicates that the proposed framework performed better in both evaluation and prediction. Moreover, the performance of the MGF framework is easily affected by data from different organizations.

4.4.2 Defect prediction with different learning schemes

In this next experiment, the twelve different learning schemes were evaluated and then used to predict defect prone-modules across the same 17 data sets. For each data set, 10 simulations of the evaluation and prediction process were performed. In each simulation, 90% of the data set was taken as *historical* data, with the remaining 10% used as *new* or unseen data.

Tables 3, 4, 5 and 6 show the evaluation and prediction performances for the 12 different learning schemes with the 17 data sets in terms of *balance* and AUC. Both the *balance* and AUC are the means across the 10 simulations for each learning scheme on each data set. For each data set, the best Evaluation and Prediction scores are shown in bold.

From Tables 3 and 4 we observe that

1) All learning schemes that performed best in evaluation also obtained best *balance*s in prediction except for the MC2 and AR1 data set. However, even for these two exceptions (NB+Log+FS and NB+Log+BE) resulted in the second-best in prediction. This suggests that the proposed framework worked well.
2) No learning scheme dominates, i.e. always outperformed the others for all the 17 data sets. This means we should choose different learning schemes for different data sets.
3) The mean evaluation *balance*s of the learning schemes NB+None+FS and NB+None+BE have

TABLE 3
Evaluation *balance* comparison of the 12 different learning schemes

Data	Naïve Bayes				J48				OneR			
	Log		None		Log		None		Log		None	
	FS	BE	FS	BE	FS	BE	FS	BE	FS	BE	FS	BE
CM1	**67.9**	66.4	50.2	50.6	31.2	38.4	30.6	38.9	33.3	33.7	33.3	33.7
KC3	67.9	**73.3**	60.6	55.8	39.3	47.7	39.3	46.3	37.4	38.4	37.5	38.4
KC4	65.7	68.9	64.8	55.8	76.5	76.6	**76.7**	76.5	68.0	68.5	68.0	68.6
MW1	61.8	64.4	**66.4**	62.8	39.1	43.6	39.1	43.1	38.7	35.2	35.8	34.6
PC1	66.0	**67.2**	50.2	49.0	41.5	47.1	41.1	47.1	37.7	35.0	36.7	35.0
PC2	**75.4**	74.0	46.9	50.8	29.3	29.3	29.3	29.6	29.3	29.3	29.3	29.3
PC3	**73.5**	72.3	72.2	73.0	30.1	43.0	30.1	43.1	33.9	34.1	33.9	34.1
PC4	**81.7**	81.0	77.0	75.9	58.3	64.3	59.1	62.8	48.2	48.3	48.2	48.3
KC1	69.9	**70.7**	62.4	57.0	47.5	48.6	47.6	48.9	42.4	42.8	42.4	42.8
MC1	80.8	**82.0**	72.9	71.4	48.2	49.1	48.5	49.5	47.9	46.5	47.9	46.5
MC2	**61.0**	59.3	57.3	53.0	56.1	53.4	55.8	53.3	48.9	46.8	49.6	47.7
JM1	59.3	**59.6**	46.4	46.3	43.5	45.6	43.3	45.8	37.7	37.5	37.7	37.5
PC5	89.1	**89.4**	85.8	79.6	60.3	62.2	60.0	61.7	47.7	48.4	47.7	48.5
AR1	31.4	**43.8**	37.6	43.6	30.4	34.4	30.4	33.9	31.5	31.5	31.5	31.5
AR3	53.7	**59.7**	56.4	55.3	54.8	53.2	54.8	53.2	52.6	53.7	52.8	54.0
AR4	66.7	64.1	57.2	58.3	70.9	58.3	**71.0**	58.2	57.0	53.7	57.1	53.7
AR6	40.4	**52.0**	48.5	45.2	35.1	38.5	35.0	38.4	38.0	35.9	38.0	35.9

TABLE 4
Prediction *balance* comparison of the 12 different learning schemes

Data	Naïve Bayes				J48				OneR			
	Log		None		Log		None		Log		None	
	FS	BE	FS	BE	FS	BE	FS	BE	FS	BE	FS	BE
CM1	**67.8**	66.9	53.3	52.7	29.3	42.2	29.3	37.9	32.1	30.7	32.1	30.7
KC3	71.3	**71.6**	57.5	52.4	44.8	47.2	48.3	52.1	37.4	39.1	37.4	39.1
KC4	73.7	69.3	68.9	58.0	**80.7**	**80.7**	**80.7**	**80.7**	69.1	70.2	69.1	70.2
MW1	61.6	61.3	**64.0**	**64.0**	42.8	42.8	42.8	38.1	38.6	41.0	38.6	41.0
PC1	67.5	**69.1**	50.7	49.8	44.5	45.4	44.6	45.4	40.4	37.0	40.4	37.0
PC2	**76.9**	69.8	49.3	52.7	29.3	29.3	29.3	29.3	29.3	29.3	29.3	29.3
PC3	**74.8**	74.2	71.5	73.4	33.2	44.2	31.0	46.0	35.9	35.9	35.9	35.9
PC4	**83.0**	81.3	76.7	74.6	60.5	62.3	58.7	63.9	49.5	48.8	49.5	48.8
KC1	70.1	**70.8**	61.6	57.3	47.6	51.6	47.9	49.9	45.5	46.1	45.5	46.1
MC1	80.4	**82.8**	72.4	69.5	47.1	48.1	47.1	48.1	49.2	47.1	49.2	47.1
MC2	64.5	61.4	**65.4**	53.2	53.8	60.6	53.8	54.5	44.1	42.7	44.1	42.7
JM1	59.3	**59.6**	46.2	46.4	43.5	45.5	43.9	45.9	37.3	37.6	37.3	37.6
PC5	89.4	**89.7**	86.3	81.2	58.1	62.1	58.1	61.4	46.6	47.0	46.7	47.0
AR1	36.5	45.7	40.6	**54.9**	29.2	29.2	29.2	29.2	29.2	29.1	29.2	29.1
AR3	60.3	**68.9**	57.5	63.2	57.3	50.3	57.3	50.3	49.1	49.1	49.1	49.1
AR4	63.1	66.6	53.1	62.9	**73.6**	57.3	**73.6**	57.3	68.1	52.5	68.1	52.5
AR6	39.2	**49.8**	43.1	36.1	28.9	36.2	28.9	36.2	32.6	32.6	32.6	32.6

been improved by schemes NB+Log+FS and NB+Log+BE by 9.73% and 16.78%, respectively. The mean prediction *balance*s of the learning schemes NB+None+FS and NB+None+BE have been improved by schemes NB+Log+FS and NB+Log+BE by 11.85% and 15.59%, respectively. It means that the Log data preprocessor indeed improved the *balance*s of the Naïve Bayes learning algorithm with both FS and BE. However, The mean evaluation *balance*s of the learning schemes J48+None+FS and J48+None+BE have been improved by schemes J48+Log+FS and J48+Log+BE by 0% and 0.41%, respectively. The mean prediction *balance*s of the learning schemes J48+None+FS and J48+None+BE have been improved by schemes J48+Log+FS and J48+Log+BE by 0% and 1.03%, respectively. It suggest that the Log data preprocessor added little to the J48 learner. Furthermore, the mean evaluation *balance* of the learning scheme OneR+None+FS has been improved by the scheme OneR+Log+FS by just 0.23%, while the mean evaluation *balance* of the learning scheme OneR+None+BE has been reduced by the scheme OneR+Log+BE by 0.24%. And the mean prediction *balance*s of the learning schemes OneR+None+FS and OneR+None+BE are the same as OneR+Log+FS and OneR+Log+BE, showing that the Log data preprocessor added nothing to the OneR learner.

This reveals that a data preprocessor can play different roles with different learning algorithms. Thus we can't judge whether a data preprocessor improves performance or not separately.

4) For both the evaluation and prediction, the *balance*s of NB+Log+FS are comparable with those of NB+Log+BE. On the other hand, the mean evaluation and prediction *balance*s of NB+None+FS are higher than those of scheme NB+None+BE, respectively. Moreover, for both the evaluation and prediction, the mean *balance*s of schemes OneR+Log+FS and OneR+None+FS are higher than those of schemes OneR+Log+BE and OneR+None+BE, respectively. On the contrary, for both the evaluation and prediction, the mean *balance*s of schemes J48+Log+FS and J48+None+FS are lower than those of schemes J48+Log+BE and J48+None+BE, respectively.
   This reveals that different attribute selectors can be suitable for different learning algorithms. Thus we cannot separately determine which attribute selectors performed better, as there is intrinsic relationship between a learning algorithm and an attribute selector.
5) For both the evaluation and prediction, the *balance*s of schemes NB+Log+FS, NB+Log+BE, NB+None+FS and NB+None+BE are much better than those of schemes J48+Log+FS, J48+Log+BE, J48+None+FS and J48+None+BE, respectively. And for both the evaluation and prediction, the mean *balance*s of schemes J48+Log+FS, J48+Log+BE, J48+None+FS and J48+None+BE are much better than that of schemes OneR+Log+FS, OneR+Log+BE, OneR+None+FS and OneR+None+BE, respectively.

From Tables 5 and 6 we observe that

1) The learning schemes that performed best in evaluation also obtained best AUCs in prediction on 10 of the 17 data sets. However, even for the 7 exceptions, the schemes which performed best in prediction also obtained the second-best in evaluation. This indicates that the proposed framework worked well.
2) No learning scheme dominates, i.e. always outperformed the others for all the 17 data sets. This means we should choose different learning schemes for different data sets.
3) The mean evaluation AUCs of the learning schemes NB+None+FS and NB+None+BE have been improved by schemes NB+Log+FS and NB+Log+BE by 0.89% and 0.25%, respectively. The mean prediction AUCs of the learning schemes NB+None+FS and NB+None+BE have been improved by schemes NB+Log+FS and NB+Log+BE by 0.51% and 0.13%, respectively. It means that the data preprocessor Log indeed improved the AUCs of the Naïve Bayes learning algorithm with both FS and BE, although it is not so much. Moreover, the mean evaluation AUCs of the learning schemes J48+None+FS and J48+None+BE have been improved by schemes J48+Log+FS and J48+Log+BE by 0.43% and 0%, respectively. The mean prediction AUC of the learning scheme J48+None+FS has been reduced by the scheme J48+Log+FS by 0.71%, and the mean prediction AUC of the scheme J48+None+BE have been improved by the scheme J48+Log+BE by 1.33%. It means that the data preprocessor Log contributed less to the J48 learner. Also the mean evaluation AUC of the learning scheme OneR+None+FS is the same with that of the scheme OneR+Log+FS, while the mean evaluation AUC of the learning scheme OneR+None+BE has been reduced by the scheme OneR+Log+BE by 0.17%. And the mean prediction AUCs of the learning schemes OneR+None+FS and OneR+None+BE are the same with that of OneR+Log+FS and OneR+Log+BE respectively. That is to say, the data preprocessor Log also contributed nothing to the OneR learner.
   This reveals that a data preprocessor can play different roles with different learning algorithms. Thus we cannot independently judge whether a data preprocessor improves performance or not.
4) The mean evaluation AUCs of the learning schemes NB+Log+FS and NB+None+FS are lower than those of schemes NB+Log+BE and NB+None+BE. The mean prediction AUC of the scheme NB+Log+FS is higher than that of NB+Log+BE. And the mean prediction AUCs of NB+None+FS and NB+None+BE are the same. However, for both the evaluation and prediction, the mean AUCs of schemes J48+Log+FS and J48+None+FS are higher than those of schemes J48+Log+BE and J48+None+BE, respectively. For both the evaluation and prediction, the mean AUCs of schemes OneR+Log+FS and OneR+None+FS are higher than those of schemes OneR+Log+BE and OneR+None+BE, respectively.
   This reveals that different attribute selectors can be suitable to different learning algorithms. Thus we cannot separately determine which attribute selectors performed better, as there is intrinsic relationship between a learning algorithm and an attribute selector.
5) For both the evaluation and prediction, the AUCs of schemes NB+Log+FS, NB+Log+BE, NB+None+FS and NB+None+BE are much better than those of schemes J48+Log+FS, J48+Log+BE, J48+None+FS and J48+None+BE, respectively. And for both the evaluation and prediction, the mean AUCs of schemes J48+Log+FS, J48+Log+BE, J48+None+FS and J48+None+BE are much better than that of schemes OneR+Log+FS, OneR+Log+BE, OneR+None+FS and OneR+None+BE, respectively.

This indicates that Naïve Bayes performs much better than J48 and J48 is better than OneR in the given

TABLE 5
Evaluation AUC comparison of the 12 different learning schemes

Data	Naïve Bayes				J48				OneR			
	Log		None		Log		None		Log		None	
	FS	BE	FS	BE	FS	BE	FS	BE	FS	BE	FS	BE
CM1	0.781	0.777	0.770	**0.788**	0.653	0.615	0.650	0.615	0.514	0.524	0.514	0.524
KC3	**0.830**	**0.830**	0.810	0.813	0.669	0.580	0.674	0.606	0.547	0.559	0.547	0.559
KC4	**0.812**	0.811	0.787	0.792	0.787	0.764	0.790	0.765	0.719	0.713	0.720	0.717
MW1	**0.802**	0.787	0.775	0.769	0.598	0.629	0.597	0.621	0.551	0.529	0.551	0.529
PC1	**0.784**	**0.784**	0.764	0.768	0.762	0.716	0.759	0.720	0.555	0.536	0.555	0.536
PC2	0.870	**0.875**	0.849	0.868	0.555	0.572	0.552	0.565	0.500	0.500	0.500	0.500
PC3	0.807	0.810	**0.814**	0.792	0.753	0.690	0.741	0.688	0.527	0.528	0.527	0.528
PC4	**0.900**	**0.900**	0.875	0.873	0.851	0.802	0.853	0.799	0.624	0.625	0.624	0.625
KC1	0.788	0.790	**0.796**	0.789	0.759	0.716	0.760	0.710	0.569	0.575	0.570	0.575
MC1	**0.944**	0.942	0.927	0.933	0.878	0.843	0.871	0.842	0.628	0.627	0.628	0.627
MC2	0.671	0.687	0.700	**0.718**	0.617	0.624	0.599	0.620	0.586	0.585	0.589	0.585
JM1	**0.717**	**0.717**	0.694	0.695	0.716	0.704	0.716	0.704	0.545	0.538	0.545	0.538
PC5	**0.963**	**0.963**	0.948	0.944	0.942	0.791	0.943	0.790	0.626	0.631	0.626	0.631
AR1	**0.636**	0.605	0.627	0.630	0.580	0.583	0.580	0.585	0.501	0.506	0.501	0.506
AR3	0.687	0.705	0.684	**0.738**	0.676	0.656	0.676	0.656	0.644	0.673	0.649	0.673
AR4	0.772	**0.785**	0.769	0.781	0.649	0.654	0.631	0.668	0.691	0.640	0.691	0.640
AR6	0.674	0.700	0.724	**0.746**	0.517	0.547	0.519	0.546	0.530	0.528	0.530	0.528

TABLE 6
Prediction AUC comparison of the 12 different learning schemes

Data	Naïve Bayes				J48				OneR			
	Log		None		Log		None		Log		None	
	FS	BE	FS	BE	FS	BE	FS	BE	FS	BE	FS	BE
CM1	0.781	0.767	0.765	**0.786**	0.638	0.630	0.671	0.599	0.509	0.500	0.509	0.500
KC3	0.827	**0.835**	0.813	0.811	0.693	0.536	0.716	0.600	0.545	0.559	0.545	0.559
KC4	0.804	**0.815**	0.800	0.789	0.813	0.807	0.809	0.807	0.719	0.711	0.719	0.711
MW1	0.766	0.771	**0.774**	0.768	0.586	0.616	0.584	0.561	0.552	0.569	0.552	0.569
PC1	0.782	**0.797**	0.765	0.786	0.782	0.679	0.789	0.695	0.574	0.551	0.574	0.551
PC2	**0.887**	0.880	0.863	0.876	0.543	0.664	0.543	0.664	0.500	0.500	0.500	0.500
PC3	**0.814**	**0.814**	0.809	0.791	0.698	0.702	0.719	0.679	0.542	0.542	0.542	0.542
PC4	**0.905**	0.903	0.871	0.870	0.847	0.818	0.848	0.774	0.633	0.628	0.633	0.628
KC1	0.793	0.794	**0.800**	0.790	0.734	0.695	0.734	0.691	0.572	0.584	0.572	0.584
MC1	**0.943**	0.941	0.933	0.936	0.911	0.805	0.906	0.799	0.640	0.626	0.640	0.626
MC2	0.685	0.687	0.693	**0.714**	0.576	0.711	0.579	0.705	0.550	0.556	0.550	0.556
JM1	0.716	0.717	0.694	0.694	**0.718**	0.708	0.712	0.712	0.544	0.537	0.544	0.537
PC5	0.962	**0.965**	0.950	0.945	0.937	0.828	0.937	0.815	0.617	0.621	0.618	0.621
AR1	**0.782**	0.653	0.686	0.658	0.587	0.618	0.587	0.623	0.491	0.491	0.491	0.491
AR3	0.672	0.697	0.720	**0.750**	0.682	0.632	0.682	0.632	0.623	0.623	0.623	0.623
AR4	0.757	0.781	0.794	**0.799**	0.622	0.680	0.622	0.630	0.759	0.635	0.759	0.635
AR6	0.647	0.646	**0.715**	0.676	0.494	0.492	0.510	0.482	0.503	0.503	0.503	0.503

context. However, from above analysis, we know that its performance is still affected by the data preprocessor and attribute selector.

For the purpose of more formally confirming whether the impacts of the data preprocessor, attribute selector and learning algorithm on the performances of both evaluation and prediction are statistically significant or not, we performed a Wilcoxon signed-rank test of medians. For all the tests, the null hypotheses are that there is no difference with ($\alpha = 0.05$). Tables 7, 8, 9, 10 and 11 show the results.

Table 7 shows the p-values of Wilcoxon signed-rank test on data preprocessors Log vs. None over the 17 data sets. From the *balance* column we can observe that for the learning algorithm Naïve Bayes with attribute selectors FS or BE, the data preprocessor Log is superior to None significantly. This means that data preprocessor Log is more suitable for Naïve Bayes. However, for the J48 and OneR learners with attribute selectors FS or BE, the preprocessor Log doesn't have significant difference with None. That means the preprocessor Log contributes little to J48 and OneR learners.

TABLE 7
p-value for the data preprocessors Log vs. None

H_a	balance	AUC
NB+FS+Log $>$ NB+FS+None	**0.0015**	**0.0001**
NB+BE+Log $>$ NB+BE+None	**0.0001**	**0.0001**
J48+FS+Log $>$ J48+FS+None	0.6855	0.0740
J48+BE+Log $>$ J48+BE+None	0.2706	0.1722
OneR+FS+Log $>$ OneR+FS+None	0.9375	0.9102
OneR+BE+Log $>$ OneR+BE+None	0.8125	0.6224

As for the performance measure of AUC, the case is similar. The preprocessor Log is more suitable for Naïve Bayes and doesn't help J48 and OneR significantly. This reveals that a data preprocessor can play different roles with different learning algorithms. Thus we can't judge whether a data preprocessor improves performance or not separately.

TABLE 8
p-value for the attribute selectors FS vs. BE

H_a	balance	AUC
NB+Log+FS > NB+Log+BE	0.9393	0.9626
NB+None+FS > NB+None+BE	**0.0401**	0.1633
J48+Log+FS > J48+Log+BE	0.9999	**0.0001**
J48+None+FS > J48+None+BE	0.9996	**0.0001**
OneR+Log+FS > OneR+Log+BE	0.5747	0.1610
OneR+None+FS > OneR+None+BE	0.5747	0.1531

Table 8 shows the *p*-values of Wilcoxon signed-rank test on attribute selectors FS vs. BE over the 17 data sets. Taking the *balance* column into account first, it can be seen that for the learning algorithm Naïve Bayes with data preprocessors None, Forward Selection is significantly superior to Backward Elimination; and for Naïve Bayes with data preprocessor Log, the difference is not significant. This means Forward Selection is more suitable for Naïve Bayes with the data preprocessor None. However, for the learning algorithm J48 with data preprocessors None or Log, Forward Selection is significantly worse than Backward Elimination. This means that Backward Elimination is more suitable for J48. Also, for the OneR learners with preprocessors None or Log, there is no significant difference between Forward Selection and Backward Elimination.

Taking the AUC column into account instead, the case is different. For the learning algorithm Naïve Bayes with data preprocessors Log, Forward Selection is significantly worse than Backward Elimination; and for Naïve Bayes with data preprocessor None, the difference is not significant. This means Backward Elimination is more suitable for Naïve Bayes with data preprocessors Log. However, for the learning algorithm J48 with data preprocessors None or Log, Forward Selection is significantly superior to Backward Elimination. This means that Backward Elimination is more suitable for J48. Also, for the OneR learners with preprocessors None or Log, there is no significant difference between Forward Selection and Backward Elimination.

This reveals that different attribute selectors can be suitable to different learning algorithms. Thus we cannot separately determine which attribute selectors performed better, as there is intrinsic relationship between a learning algorithm and an attribute selector.

TABLE 9
p-value for the algorithms Naïve Bayes vs. J48

H_a	balance	AUC
Log+FS+NB > Log+FS+J48	**0.0001**	**0.0001**
Log+BE+NB > Log+BE+J48	**0.0001**	**0.0001**
None+FS+NB > None+FS+J48	**0.0001**	**0.0001**
None+BE+NB > None+BE+J48	**0.0001**	**0.0001**

Table 9 shows the *p*-values of Wilcoxon signed-rank test on learning algorithms Naïve Bayes vs. J48 over the 17 data sets. From it we observe that Naïve Bayes is superior to J48 significantly in all the cases. This means Naïve Bayes is much better than J48 on the given data sets.

TABLE 10
p-value for the algorithms Naïve Bayes vs. OneR

H_a	balance	AUC
Log+FS+NB > Log+FS+OneR	**0.0001**	**0.0001**
Log+BE+NB > Log+BE+OneR	**0.0001**	**0.0001**
None+FS+NB > None+FS+OneR	**0.0001**	**0.0001**
None+BE+NB > None+BE+OneR	**0.0001**	**0.0001**

Table 10 shows the *p*-values of Wilcoxon signed-rank test on learning algorithms Naïve Bayes vs. OneR over the 17 data sets. From it we observe that Naïve Bayes is superior to OneR significantly in all the cases. This means Naïve Bayes is much better than OneR on the given data sets.

TABLE 11
p-value for the algorithms J48 vs. OneR

H_a	balance	AUC
Log+FS+J48 > Log+FS+OneR	**0.0053**	**0.0001**
Log+BE+J48 > Log+BE+OneR	**0.0001**	**0.0001**
None+FS+J48 > None+FS+OneR	**0.0053**	**0.0001**
None+BE+J48 > None+BE+OneR	**0.0001**	**0.0001**

Table 11 shows the *p*-values of Wilcoxon signed-rank test on learning algorithms J48 vs. OneR over the 17 data sets. From it we observe that J48 is superior to OneR significantly in all the cases. This means J48 is much better than OneR on the given data sets.

To summarize, since no learning scheme dominates, we should choose different learning schemes for different data sets. Moreover, different elements of a learning scheme play different roles. Each of data preprocessor, learning algorithm, or attribute selector is just one element of the learning scheme, which should not be evaluated separately. On the contrary, a learning scheme should be evaluated as holistically.

5 Conclusion

In this paper, we have presented a novel benchmark framework for software defect prediction. The framework involves evaluation and prediction. In the evaluation stage, different learning schemes are evaluated and the best one is selected. Then in the prediction stage, the best learning scheme is used to build a predictor with all historical data and the predictor is finally used to predict defect on the new data.

We have compared the proposed framework with MGF's study [23], and pointed out the potential bias in their baseline experiment. We have also performed a baseline experiment to simulate the whole process of defect prediction in both MGF's study and our framework. From our experimental results we observe that there is a bigger difference between the evaluation performance and the actual prediction performance in MGF's study than with our framework. This means that the results that they report are over optimistic. Whilst this might seem like some small technicality the impact is profound. When we perform statistical significance testing we find dramatically different findings and that are highly statistically significant but in *opposite* directions. The real point is not which learning scheme does 'better' but how should one set about answering this question. From our experimental results we also observe that the predictions of Menzies et al. on the AR data are much more biased than that on the NASA data and the performance of the MGF framework is varying greatly with data from different resources. Thus, we contend our framework is less biased and more capable of yielding results closer to the 'true' answer. Moreover, our framework is more stable.

We have also performed experiments to explore the impacts of different elements of a learning scheme on the evaluation and prediction. From these results we see that a data preprocessor/attribute selector can play different roles with different learning algorithms for different data sets, and that no learning scheme dominates, i.e. always outperforms the others for all data sets. This means we should choose different learning schemes for different data sets and consequently the evaluation and decision process is important.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers and Dr. Tim Menzies for their helpful comments. This work is supported by the National Natural Science Foundation of China under grants 90718024 and 61070006.

REFERENCES

[1] B. T. Compton and C. Withrow, "Prediction and control of ada software defects," *J. Syst. Softw.*, vol. 12, no. 3, pp. 199–207, 1990.

[2] J. Munson and T. M. Khoshgoftaar, "Regression modelling of software quality: empirical investigation," *J. Electron. Mater.*, vol. 19, no. 6, pp. 106–114, 1990.

[3] N. B. Ebrahimi, "On the statistical analysis of the number of errors remaining in a software design document after inspection," *IEEE Trans. Softw. Eng.*, vol. 23, no. 8, pp. 529–532, 1997.

[4] S. Vander Wiel and L. Votta, "Assessing software designs using capture-recapture methods," *IEEE Trans. Softw. Eng.*, vol. 19, no. 11, pp. 1045–1054, 1993.

[5] P. Runeson and C. Wohlin, "An experimental evaluation of an experience-based capture-recapturemethod in software code inspections," *Empirical Softw. Eng.*, vol. 3, no. 4, pp. 381–406, 1998.

[6] L. C. Briand, K. El Emam, B. G. Freimut, and O. Laitenberger, "A comprehensive evaluation of capture-recapture models for estimating software defect content," *IEEE Trans. Softw. Eng.*, vol. 26, no. 6, pp. 518–540, 2000.

[7] K. El Emam and O. Laitenberger, "Evaluating capture-recapture models with two inspectors," *IEEE Trans. Softw. Eng.*, vol. 27, no. 9, pp. 851–864, 2001.

[8] C. Wohlin and P. Runeson, "Defect content estimations from review data," in *ICSE '98: Proceedings of the 20th International Conference on Software engineering*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 400–409.

[9] G. Q. Kenney, "Estimating defects in commercial software during operational use," *IEEE Trans. Reliability*, vol. 42, no. 1, pp. 107–115, 1993.

[10] P. F., T. Ragg, and R. Schoknecht, "Using machine learning for estimating the defect content after an inspection," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 17–28, 2004.

[11] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 675–689, 1999.

[12] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software defect association mining and defect correction effort prediction," *IEEE Trans. Softw. Eng.*, vol. 32, no. 2, pp. 69–82, 2006.

[13] A. Porter and R. Selby, "Empirically guided software development using metric-based classification trees," *IEEE Software*, vol. 7, pp. 46–54, 1990.

[14] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Trans. Softw. Eng.*, vol. 18, no. 5, pp. 423–433, 1992.

[15] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, 1996.

[16] T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, and S. J. Aud, "Application of neural networks to software quality modeling of a very large telecommunications system," *IEEE Trans. Neural Networks*, vol. 8, no. 4, pp. 902–909, 1997.

[17] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Classification tree models of software quality over multiple releases," in *Proceedings of the 10th International Symposium on Software Reliability Engineering*, Washington, DC, USA, 1999, p. 116.

[18] K. Ganesan, T. M. Khoshgoftaar, and E. Allen, "Case-based software quality prediction," *Int'l J. Software Eng. and Knowledge Eng.*, vol. 10, no. 2, pp. 139–152, 2000.

[19] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "Comparing case-based reasoning classifiers for predicting high risk software components," *J. Syst. Softw.*, vol. 55, no. 3, pp. 301–320, 2001.

[20] T. M. Khoshgoftaar and N. Seliya, "Fault prediction modeling for software quality estimation: Comparing commonly used techniques," *Empirical Softw. Engg.*, vol. 8, no. 3, pp. 255–283, 2003.

[21] ——, "Analogy-based practical classification rules for software quality estimation," *Empirical Softw. Engg.*, vol. 8, no. 4, pp. 325–350, 2003.

[22] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, Washington, DC, USA, 2004, pp. 417–428.

[23] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, 2007.

[24] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, 2008.

[25] I. Myrtveit, E. Stensrud, and M. Shepperd, "Reliability and validity in comparative studies of software prediction models," *IEEE Trans. Softw. Eng.*, vol. 31, no. 5, pp. 380–391, 2005.

[26] K. Srinivasan and D. Fisher, "Machine learning approaches to estimating development effort," *IEEE Trans. Softw. Eng.*, vol. 21, no. 2, pp. 126–137, 1995.

[27] J. Tian and M. Zelkowitz, "Complexity measure evaluation and selection," *IEEE Trans. Softw. Eng.*, vol. 21, pp. 641–649, 1995.

[28] M. A. Hall, "Correlation based attribute selection for discrete and numeric class machine learning," in *17th International Conf. on Mach. Learning*, 2000, pp. 359–366.

[29] S. Wagner, "A literature survey of the quality economics of defect-detection techniques," in *ACM/IEEE International Symposium on Empirical Software Engineering*, Rio de Janeiro, 2006, pp. 194–203.

[30] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling, "What do we know about defect detection methods?" *IEEE Software*, vol. 23, no. 3, pp. 82–90, 2006.

[31] J. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[32] M. Hall and G. Holmes, "Benchmarking attribute selection techniques for discrete class data mining," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 6, pp. 1437–1447, 2003.

[33] C. Cardie, "Using decision trees to improve case-based learning," in *10th International Conference on Machine Learning*, Amherst, MA, 1993, pp. 25–32.

[34] M. Kubat, D. Flotzinger, and G. Pfurtscheller, "Discovering patterns in eeg-signals: Comparative study of a few methods," in *European Conference on Machine Learning ECML'93*, Vienna, 1993, pp. 366–371.

[35] D. Kibler and D. W. Aha, "Learning representative exemplars of concepts: an initial case study," in *4th International Workshop on Machine Learning*, 1987, pp. 24–30.

[36] R. Kohavi and G. John, "Wrappers for feature selection for machine learning," *Artificial Intelligence*, vol. 97, pp. 273–324, 1997.

[37] M. Chapman, P. Callis, and W. Jackson, "Metrics data program," NASA IV and V Facility, Tech. Rep., 2004.

[38] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, vol. 81, no. 5, pp. 649–660, 2008.

[39] B. Turhan and A. Bener, "Analysis of naive bayes assumptions on software fault data: An empirical study," *Data Knowl. Eng.*, vol. 68, no. 2, pp. 278–290, 2009.

[40] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for evaluating fault prediction models," *Empirical Softw. Engg.*, vol. 13, pp. 561–595, 2008.

[41] L. Zhan and M. Reformat, "A practical method for the software fault-prediction," in *IEEE International Conference on Information Reuse and Integration*, 2007.

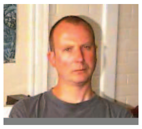
[42] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with precision: A response to comments on data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 637–640, 2007.

[43] A. Oral and A. Bener, "Defect prediction for embedded software," in *22nd International Symposium on Computer and Information Sciences*. Ankara: IEEE Computer Society, 2007, pp. 1–6.

[44] B. Turhan and A. Bener, "Software defect prediction: heuristics for weighted naive bayes," in *Proceedings of the 7th International Conference on Quality Software*, 2007, pp. 231–237.

Qinbao Song received the Ph.D. degree in computer science from Xi'an Jiaotong University, Xi'an, China, in 2001. He is currently a Professor of software technology in the Department of Computer Science and Technology, Xi'an Jiaotong University, where he is also the Deputy Director of the Department of Computer Science and Technology. He is also with the State Key Laboratory of Software Engineering, Wuhan University, Wuhan, China. He has authored or coauthored more than 80 referred papers in the areas of machine learning and software engineering. He is a board member of the Open Software Engineering Journal. His current research interests include machine learning, empirical software engineering, and trustworthy software.

Zihan Jia received the BS degree in computer science from Xi'an Jiaotong University, Xi'an, China, in 2008. He is currently a master student in the Department of Computer Science and Technology, Xi'an Jiaotong University. His research focuses on software engineering data mining.



Martin Shepperd received the Ph.D. degree in computer science from the Open University in 1991. For a number of years, he was with a bank as a Software Developer. He is currently Professor of Software Technology at Brunel University, London, U.K. He has authored or coauthored more than 120 refereed papers and three books in the areas of software engineering and machine learning. From 1992 to 2007, he was the Editor-in-Chief of the journal Information & Software Technology and presently an Associate Editor of Empirical Software Engineering. Prof. Shepperd was an Associate Editor of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING (2000-2004).

Shi Ying received the Ph.D. degree in computer science from Wuhan University, Wuhan, China, in 1999. He is currently a professor of Wuhan University. He is vice dean of Computer School of Wuhan University, he is also the Deputy Director of State Key Laboratory of Software Engineering at Wuhan University. He has authored or coauthored more than 100 referred papers in the area of software engineering. His current research interests include service-oriented software engineering, semantic web service, and trustworthy software.

Jin Liu received the Ph.D. degree in computer science from the State Key Lab of Software Engineering, Wuhan University, Wuhan, China, in 2005. He is currently an Associate Professor in the State Key Lab of Software Engineering, Wuhan University. He has authored or coauthored a number of research papers in international journals. His research interests include software modeling on the Internet and intelligent information processing.