International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 2 Issue 7, July - 2013

A Systematic Analysis Of Fault-Prone Prediction In A Software Framework

M. Venkata Prasad Rao, M.Tech- Software Engg, SVCET, Chittoor-AP.
G.N. Vivekananda, M.Tech. Assist. Professor, SVCET, Chittoor –AP,

Abstract

BACKGROUND - The accurate prediction of where faults are likely to occur in code can help test effort, reduce costs, and improve the quality of software. Predicting fault – prone software component is an economically important activity and so has received a good deal of attention. OBJECTIVE: We proposed and evaluate a systematic analysis of software framework for software fault prediction that supports (1) unbiased (2) Comprehensive cooperation between competing prediction system. METHOD: The framework is comprised of (1) Design evaluation (2) Fault prediction components. The design evaluation analyzes the prediction performance of compacting learning designs for given historical data sets. The fault prediction builds models according to the evaluated learning design and predicts software faults with new data according to the constructed model. RESULTS : The result shows that we should choose different learning designs for different data sets, that small details in conducting how evolution are conducted can completely reverse findings and last, that our proposed frame work is more effective and less prone to bais then previous approaches. CONCLUSTIONS-Failure to properly or fully evaluate a learning design can be misleading; however, these problems may be overcome by our proposed framework.

1. Introduction

Software fault prediction has been an important research topic in the software engineering filed for more than 32 years. Our analysis investigate how model performance in affected by the context in which the model was developed, the independent variables used in the model, and the technique on which the model was build.

Fault prediction modelling is an important area of reach and the subject of many previous studies these studies typically produce fault prediction models which allow software engineers to focus development activities on fault-prone code, thereby improving software quality and making better use of resources.

The current fault prediction work focuses on (1) estimating the number of faults remaining in software systems. (2) Discovering faults association and (3) classifying the fault- prone of software component. The first type of work employs statically approaches, Capture- Recapture(CR) model and detection profile methods(DPM) to estimate the number of faults 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 and can be used to control the software process and grange the likely delivered quality of the a software system. The second type of work borrows association rule mining algorithms from the data mining community of revel software fault associations which can be used for these purposes. First, finding as many related faults as possible to the detected faults and consequently make more effective correction to the software. This may be useful as it permits more directed testing and more effective use of limited testing resource. Second, helping evaluate reviewers results deriving our inspection. Thus , a recommendation might be that his /her work should be re-inspected for completeness. Third, assisting managers in improving the software process through analysis of the reasons why some faults frequently occur together. If the analysis leads to the identification of a process problem managers can device corrective action. The third type of work classifies software components as fault-prone and non-fault-proneby means of metric based classification. Being able to predict which component s are more likely to be fault-prone supports better targeted testing resources and therefore improved efficiency.

2. Related Works

Much research on detection of fault prone software modules has been carried out so far. Previous studies can be categorized by data sets, metrics and classification of techniques. Software metrics related to program attribute such as lines of code, complexity frequency of modification, coherency, coupling, and so on are used in most of previous studies. In those studies, such metrics are considered as explanatory variables and fault proneness are considered as objective variables. Then mathematical models are constructed from those metrics. The selection of metrics varies according to studies. For example, studies such as used NASA MDP collection metrics. The object oriented metrics are used in. Some studies used metrics based on metrics collection tools.

Menzies, Greenwald, and Frank (MGF) published a study in this journal in 2007 in which they compared the performance of two machine learning techniques (Rule Induction and Naive Bayes) to predict software components containing faults. To do this, they used the NASA MDP repository, which, at the time of their research, contained 10separate 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 fault 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."
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 concluded that the forward selection (FS) search was well suited to Naive Bayes but the backward elimination (BE) search is more suitable for C4.5. Cardie found using a decision tree to select attributes helped the nearest neighbor algorithm to reduce its prediction error. Kubat et al. used a decision tree filtering attributes for use with a Naive Bayesian classifier and obtained a similar result. However, Kibler and Aha 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 preprocessing, and attribute selection method, not merely one or two of them.
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 x 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. These seemingly minor issues motivate the development of our general-purpose fault 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: design evaluation and fault prediction. The design evaluation focuses on evaluating the performance of a learning design, while the fault prediction focuses on building a final predictor using historical data according to the learning design and after which the predictor is used to predict the fault-prone components of a new (or unseen) software system.

A learning design is comprised of:

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 the following: 1) We choose the entire learning design, not just one out of the learning algorithm, attribute selector, or data preprocessor;2) we use the appropriate data to evaluate the performance of a design. That is, we build a predictive model according to a design 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 Framework

3. 1 *Overview of Software Framework*

Generally, before building fault prediction model(s) and using them for prediction purposes, we first need to decide which learning design should be used to construct the model. Thus, the predictive performance of the learning design(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 fault prediction framework that provides guidance to address these potential shortcomings. The framework consists of two components: 1) design evaluation and 2) fault prediction. Fig. 1 contains the details. At the design evaluation stage, the performances of the different learning designs are evaluated with historical data to determine whether a certain learning design performs sufficiently well for prediction purposes or to select the best from a set of competing designs. 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 designs, 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 design and to further determine whether or not to apply the learning design or select one best design from the given designs. At the fault prediction stage, according to the performance report of the first stage, a learning design is selected and used to build a prediction model and predict software fault. From Fig. 1, we observe that all of 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 fault-proneness of new software components.MGF proposed a baseline experiment and reported the performance of the Naive Bayes data miner with log-filtering as well as attribute selection, which performed the design 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.

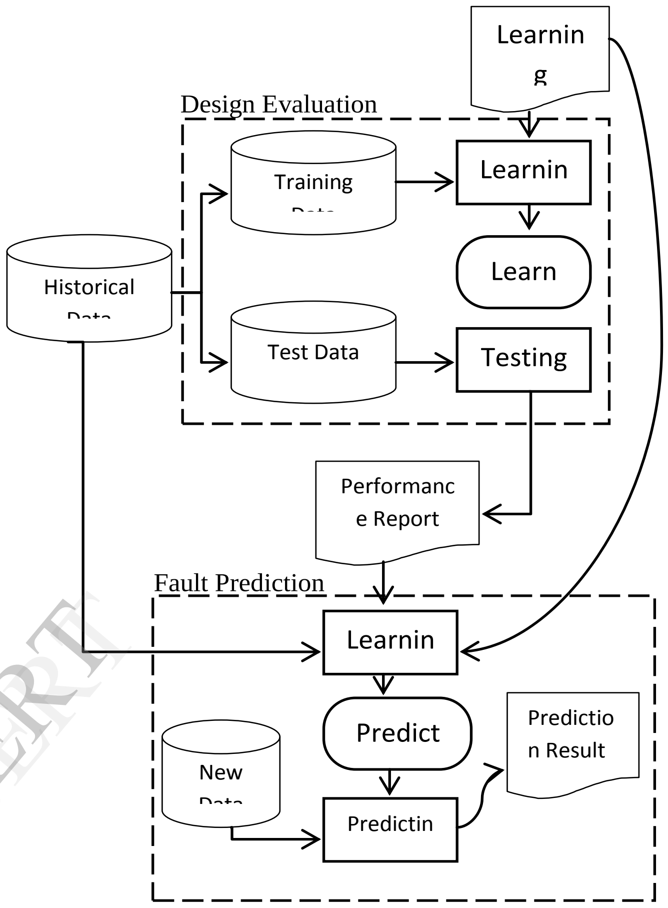

Fig 1. Proposed Software fault prediction Framework

3.2 *Design Evaluation*

The design evaluation is a fundamental part of the software fault prediction framework. At this stage, different earning designs are evaluated by building and evaluating learners with them. The first problem of design 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, an M x 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 x N models are built in all during the period of evaluation; thus M x N results are obtained on each data set about the performance of the each learning design. After the training-test splitting is done each round, both the training data and learning design(s) are used to build a learner. A learning design 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 art of building a practical learner. In this step, the training data are preprocessed, such as removing outliers, handling missing values, and discretizing or transforming numeric attributes. In our experiment, we use a log-filtering preprocessor which replaces all numeric's n with their logarithms *ln(n)*, such as used in MGF.
2. *Attribute selection*. The data sets may not have originally been intended for fault prediction; thus, even if all of the attributes are useful for its original task, not all may be helpful for fault prediction. Therefore, attribute selection has to be performed on the training data. Attribute selection methods can be categorized as either filters or wrappers. 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 the most use of the data, we use an M x 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 x N passes of such validation.

Pseudo code

This is the detailed design evaluation process is described with the pseudo code which consists of function *Learning* and function *AttrSelect*. The function *Learning* is used to build a learner with a given learning design, and the function *AttrSelect* performs attribute selection with a learning algorithm.

```
Function Learning(data, design)
Input :data - the data on which the learner is
built;
design - the learning design.
Output:learner - the final learner built on data
        withdesign;
        bestAttrs - the best attribute subset
selected bythe attribute selector of design
1 m=10; /*number of repetitions for
attributeselection * /
2 n= 10;/* number of folds for attribute
selection*/
3 d= Preprocessing(data; design.preprocessor);
4 bestAttrs=AttrSelect(d, design. Algorithm,
Design.attrSelector, m, n);
5 d'= selectbestAttrsfrom d;
6 learner=BuildClassifier(d',design.algorithm);
/* build a classifier on d' with the learning
algorithm of design */
```

3.3 Fault Prediction

The fault prediction part of our framework is straightforward; it consists of predictor construction and fault prediction.

During the period of the predictor construction:

1. A learning design is chosen according to the Performance Report.
2. A predictor is built with the selected learning design and the whole historical data. While evaluating a learning design, 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 of 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 fault with preprocessed new data.

The detailed fault prediction process is described with pseudo code in the following Procedure *Prediction*.

```
Procedure Prediction (historicalData, newData,
Design);
Input: historicalData - the historical data;
newData-thenew data;
Design- the learning design.
Output: Result - the predicted result for the
newData
1 [Predictor,bestAttrs]=Learning
                    (historicalData, Design);
2 d= select bestAttrsfrom newData;
3 Result=Predict(d,Predictor);
/* predict the class label of d with
Predictor  */
```

4. Experiments

We have to collect both fault-prone(FP) modules and non fault-prone(NFP) modules from source code repository for this research. The collection of such modules seems easy for a software project which has a bug database such as an Open Source Software development. However, even in such an environment, the revision control system and bug database system are usually separated and thus tracking on the fault-prone modules needs effort. In the development of software in companies, the situation becomes harder.

4.1 Data Sets

We used the data taken from the public NASA MDP repository, which was also used by MGF and many others. What's more, the AR data from the PROMISE repository where also used., Thus there are 17 data sets in total, 13 from NASA and the remaining 4 from the PROMISE repository.

Each data set is comprised of a number of software modules, each containing the corresponding number of faults and various software static code attributes. After preprocessing, modules that contain one or more faults were labelled as faultive. Besides LOC counts, the data sets include Halstead attributes, as well as McCabe complexity measures.

4.2 Performance Measures

The receiver operating characteristic(ROC) curve is often used to evaluate the performance of binary predictors. The following shown in Fig. 2. The y-axis shows probability of detection (pd) and the y-axis shows probability of false alarms(pf).

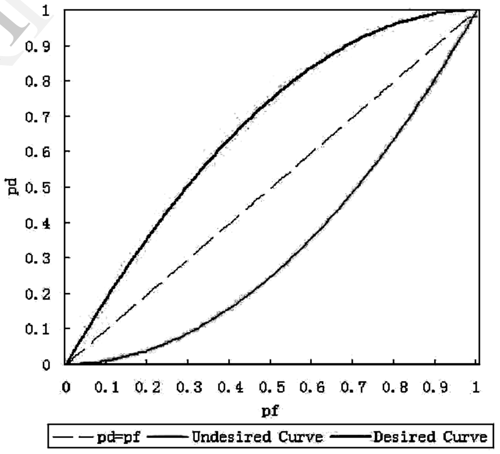

Fig 2.The ROC Curves.

Formal definitions for pd and pf are given in(1) and (2) respectively. Obviously, higher pds and lower pfs are desired. The point(pf=0,pd=1) is the ideal position where we recognize all faultive modules and never make mistakes.

$$pd = tpr = \frac{TP}{TP + FN'} \quad (1)$$

$$pf = fpr = \frac{FP}{FP + TN'} \quad (2)$$

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

MGF introduced a performance measure called balance ,which is used to choose the optimal (pd, pf) pairs. The definition is shown in (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.
Zhang and Reformat [41] argue that using (pd, pf) performance measures in the classification of imbalanced data is not practical due to low precisions. By contrast, MGF argue that precision has an unstable nature and can bemisleading 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.
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.

5. 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 design or not.

5.1FrameworkComparison

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 percent of the data as historical data, the remaining 10 percent as new data.
2. We replicated MGF's work with the historical data. First, all of the historical data were 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; . . . , Nth top-ranked attribute(s) were evaluated step by step. Each subset was evaluated by a 10 x 10-way cross-validation with the Naive 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 were processed by the log-filtering method and reduced by the selected best attribute subset and the resultant data were used to build a Naive Bayes predictor. Then, the predictor was used to predict fault with the new data that were processed by same way as that of the historical data.
3. We also simulated the whole fault prediction process presented in our framework.

In order to be comparable with MGF, were stricted our learning design to the same preprocessing method, attribute selection method, and the same learning algorithm. A 10 x 10-way cross-validation was used to evaluate the learning design. The learning design was wrapped in each validation of the 10 x 10-way cross-validation, which is different from MGF's study. Specifically, as described in the design evaluation procedure, we applied the learning design only to the training data, after which the final Naive 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 were processed according to the learning design, and a Naive Bayes predictor was built with the processed data. Then, the predictor was used to predict fault with the new data that were processed by the same way as that of the historical data.

6. Conclusions

In this paper, we have presented a novel benchmark framework for software fault prediction. The framework involves evaluation and prediction. In the evaluation stage, different learning designs are evaluated and the best one is selected. Then, in the prediction stage, the best learning design is used to build a predictor with all historical data and the predictor is finally used to predict fault on the new data. We have compared the proposed framework with MGF's study and pointed out the potential bias in their baseline experiment. We have also performed a baseline experiment to simulate the whole process of fault 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. While this might seem like some small technicality, the impact is profound. When we perform statistical significance testing, we find dramatically different findings that are highly statistically significant but in opposite directions. The real point is not which learning design 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 varies 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 design 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 design dominates, i.e., always outperforms the others for all data sets. This means we should choose different learning designs for different datasets, and consequently, the evaluation and decision processes important.

REFERENCES

[1] B.T. Compton and C. Withrow, "Prediction and Control of ADA Software Defects," J. Systems and Software, vol. 12, no. 3, pp. 199- 207, 1990.
[2] J. Munson and T.M. Khoshgoftaar, "Regression Modelling ofSoftware Quality: Empirical Investigation," J. Electronic Materials, vol. 19, no. 6, pp. 106-114, 1990.
[3] N.B. Ebrahimi, "On the Statistical Analysis of the Number ofErrors Remaining in a Software Design Document After Inspection,"IEEE Trans. Software Eng., vol. 23, no. 8, pp. 529-532, Aug. 1997.
[4] S. Vander Wiel and L. Votta, "Assessing Software Designs UsingCapture-Recapture Methods," IEEE Trans. Software Eng., vol. 19, no. 11, pp. 1045-1054, Nov. 1993.
[5] P. Runeson and C. Wohlin, "An Experimental Evaluation of anExperience-Based Capture-Recapture Method in Software CodeInspections," Empirical Software Eng., vol. 3, no. 4, pp. 381-406,1998.
[6] L.C. Briand, K. El Emam, B.G. Freimut, and O. Laitenberger, "AComprehensive Evaluation of Capture-Recapture Models forEstimating Software Defect Content," IEEE Trans. Software Eng.,vol. 26, no. 6, pp. 518-540, June 2000.
[7] K. El Emam and O. Laitenberger, "Evaluating Capture-RecaptureModels with Two Inspectors," IEEE Trans. Software Eng., vol. 27,no. 9, pp. 851-864, Sept. 2001.
[8] C. Wohlin and P. Runeson, "Defect Content Estimations fromReview Data," Proc. 20th Int'l Conf. Software Eng., pp. 400-409,1998.
[9] G.Q. Kenney, "Estimating Defects in Commercial Software duringOperational Use," IEEE Trans. Reliability, vol. 42, no. 1, pp. 107- 115, Mar. 1993.
[10] F. Padberg, T. Ragg, and R. Schoknecht, "Using Machine Learningfor Estimating the Defect Content After an Inspection," IEEETrans. Software Eng., vol. 30, no. 1, pp. 17-28, Jan. 2004.
[11] N.E. Fenton and M. Neil, "A Critique of Software DefectPrediction Models," IEEE Trans. Software Eng., vol. 25, no. 5,pp. 675-689, Sept./Oct. 1999.
[12] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "SoftwareDefect Association Mining and Defect Correction Effort Prediction,"IEEE Trans. Software Eng., vol. 32, no. 2, pp. 69-82, Feb. 2006.

[13] A. Porter and R. Selby, “Empirically Guided Software DevelopmentUsing Metric-Based Classification Trees,” IEEE Software,vol. 7, no. 2, pp. 46-54, Mar. 1990.

[14] J.C. Munson and T.M. Khoshgoftaar, “The Detection of Fault-Prone Programs,” IEEE Trans. Software Eng., vol. 18, no. 5, pp. 423-433, May 1992.

[15] V.R. Basili, L.C. Briand, and W.L. Melo, “A Validation of Object-Oriented Design Metrics as Quality Indicators,” IEEE Trans.Software Eng., vol. 22, no. 10, pp. 751-761, Oct. 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. NeuralNetworks, vol. 8, no. 4, pp. 902-909, July 1997.

[17] T.M. Khoshgoftaar, E.B. Allen, W.D. Jones, and J.P. Hudepohl,“Classification Tree Models of Software Quality over MultipleReleases,” Proc. 10th Int’l Symp.Software Reliability Eng., pp. 116-125, 1999.

[18] K. Ganesan, T.M. Khoshgoftaar, and E. Allen, “Case-BasedSoftware Quality Prediction,” Int’l J. Software Eng. and KnowledgeEng., 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. Systems and Software, vol. 55, no. 3, pp. 301-320,2001.

[20] L. Zhan and M. Reformat, “A Practical Method for the SoftwareFault-Prediction,” Proc. IEEE Int’l Conf. Information Reuse andIntegration, pp. 659-666, 2007.