Empir Software Eng (2008) 13:561–595
DOI 10.1007/s10664-008-9079-3

Techniques for evaluating fault prediction models

Yue Jiang · Bojan Cukic · Yan Ma

Published online: 12 August 2008

Editor: Tim Menzies

Abstract Many statistical techniques have been proposed to predict fault-proneness of program modules in software engineering. Choosing the "best" candidate among many available models involves performance assessment and detailed comparison, but these comparisons are not simple due to the applicability of varying performance measures. Classifying a software module as fault-prone implies the application of some verification activities, thus adding to the development cost. Misclassifying a module as fault free carries the risk of system failure, also associated with cost implications. Methodologies for precise evaluation of fault prediction models should be at the core of empirical software engineering research, but have attracted sporadic attention. In this paper, we overview model evaluation techniques. In addition to many techniques that have been used in software engineering studies before, we introduce and discuss the merits of cost curves. Using the data from a public repository, our study demonstrates the strengths and weaknesses of performance evaluation techniques and points to a conclusion that the selection of the "best" model cannot be made without considering project cost characteristics, which are specific in each development environment.

Keywords Fault-prediction models · Model evaluation · Predictive models in software engineering · Empirical studies

1 Introduction

Early detection of fault-prone software components enables verification experts to concentrate their time and resources on the problem areas of the system under development.

Y. Jiang · B. Cukic (✉)
Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV 26506-6109, USA
e-mail: bojan.cukic@mail.wvu.edu

Y. Ma
Department of Statistics, West Virginia University, Morgantown, WV 26506-6109, USA

The ability of software quality models to accurately identify critical components allows for the application of focused verification activities ranging from manual inspection to testing, static and dynamic analysis, and automated formal analysis methods. Software quality models, thus, help ensure the reliability of the delivered products. It has become imperative to develop and apply good software quality models early in the software development life cycle, especially in large-scale development efforts.

Most fault prediction techniques rely on historical data. Experiments suggest that a module currently under development is fault-prone if it has the same or similar properties, measured by software metrics, as a faulty module that has been developed or released earlier in the same environment (Khoshgoftaar et al. 1997). Therefore, historical information helps us to predict fault-proneness. Many modeling techniques have been proposed for and applied to software quality prediction. Some of these techniques, logistic regression (Basili et al. 1996) for example, aim to use domain-specific knowledge to establish the input (software metrics) output (software fault-proneness) relationship. Some techniques, such as classification trees (Gokhale and Lyu 1997; Khoshgoftaar and Seliya 2002; Selby and Porter 1988), neural networks (Khoshgoftaar and Lanning 1995), and genetic algorithms (Azar et al. 2002), try to examine the available large-size datasets to recognize patterns and form generalizations. Some utilize a single model to predict fault-proneness; others are based on ensemble learning by building a collection of models from a single training dataset (Guo et al. 2004; Ma 2007).

A wide range of classification algorithms has been applied to different data sets. Different experimental setups result in a limited ability to comprehend algorithm's strengths and weaknesses. A modeling methodology is good if it is able to perform well on all data sets, or at least most of them. Recently, several software engineering data repositories become publicly available (Metrics Data Program NASA IV&V facility, http://mdp.ivv.nasa.gov/). Therefore, these data sets can be used to validate and compare the proposed predictive models. In order to identify reliable classification algorithms, Challagulla *et al.* recommended trying a set of different predictive models (Challagulla et al. 2005). Guo *et al.* (2004) suggested building a toolbox for software quality engineers which includes "good" prediction performers (Guo et al. 2004). But choosing the "best" model among many available ones involves model performance assessment and evaluation. In order to simplify model comparison, we should use appropriate and consistent performance measures.

Model performance comparison received attention in the software engineering literature (El-Emam et al. 2001). Nevertheless, empirical studies continue to apply different performance measures. Consequently, such studies do not encourage cross comparison with the results of work performed elsewhere. Many studies use inadequate performance metrics, those that do not reveal sufficient level of details for future comparison. For these reasons, the objectives of this paper include:

1) A survey of commonly used model performance metrics and a discussion of their merits,
2) An introduction of cost curves, a new model evaluation technique in software engineering, and
3) A comparison of model evaluation techniques and a guide to their selection.

We believe that our findings and recommendations have a potential to enhance statistical validity of future experiments and, ultimately, further the state of practice in fault prediction modeling.

1.1 Experimental Set-up

We use NASA MDP datasets (Metrics Data Program NASA IV&V facility, http://mdp.ivv.nasa.gov/), listed in Table 1, to illustrate problems in model evaluation. The metrics in these datasets describe projects which vary in size and complexity, programming languages, development processes, etc. When reporting a fault prediction modeling experiment, it is important to describe the characteristics of the datasets. Each data set contains twenty-one software metrics, which describe product's size, complexity and some structural properties (Metrics Data Program NASA IV&V facility, http://mdp.ivv.nasa.gov/). A detailed description of some of the intrinsic characteristics of NASA MDP projects is provided in Appendix A.

In the illustrative examples, we use six well known classification algorithms. To remind the reader, our goal is to demonstrate the strengths and drawbacks of specific performance measures, rather than identifying the "best" algorithm. For this reason, the choice of classifiers is orthogonal with respect to the intended contributions. The six classifiers we selected are Random Forest, Naïve Bayes, Bagging, J48, Logistic Regression and IBk. These algorithms represent a broad range of machine learning approaches. Recent studies indicate that these classifiers provide better than average performance in software fault prediction (Menzies et al. 2007). A brief description of these algorithms is offered in Appendix B. All reported experiments utilize classifier implementations from a well-known software package WEKA (Witten and Frank 2005). All performance measurements are always generated by tenfold cross-validation of classification.

The rest of this paper is organized as follows. The numeric evaluation metrics are discussed in Section 2. The graphical summarization methods are described in Section 3. Statistical inferences are often required to compare classifiers. Statistical testing techniques that compare predictive models are the subject of Section 4. Section 5 describes two fault prediction experiments in detail and demonstrates that evaluation measures frequently result in contradicting conclusions. For this reason, Section 6 summarizes the guidelines for the selection of model evaluation techniques in practice. We conclude the paper with a summary in Section 7.

Table 1 Characteristics of projects from NASA MDP (Metrics Data Program NASA IV&V facility, http://mdp.ivv.nasa.gov/), used in our experiments

Project	Language	Number of modules	Faulty (%)	Description
KC1	C++	2,109	15	Storage management, receiving/processing ground data
KC2	C++	523	21	Science data processing
KC4	Perl	125	48	Storage management for ground data
JM1	C	10,885	19	Real-time prediction ground system
PC1	C	1,109	7	Flight software for earth orbiting satellite
PC5	C++	17,186	3	A safety enhancement upgrade system for cockpit
CM1	C	498	10	A NASA spacecraft instrument
MC2	C++	161	32	A video guidance system

2 Evaluation of Predictive Models: Numerical Indices

Numerical performance evaluation indices are the most common in the software engineering literature. This section surveys them and describes their individual strengths and shortcomings.

2.1 Overall Accuracy and Error Rate

The overall accuracy measures the chance of correctly predicting the fault proneness of individual modules. It ignores the data distribution and cost information. Therefore, it can be a misleading criterion as faulty modules are likely to represent a minority of the modules in the dataset (see Table 1).

Table 2 lists the overall accuracy and the detection rates of faulty modules (denoted as *PD*, defined in the next section) for four projects: *PC1*, *KC1*, *KC2* and *JM1*. The disadvantage of overall accuracy as a performance evaluation metric is obvious. For every project in Table 2, the highest overall accuracy is accompanied with the lowest detection rate in the fault-prone class. In software engineering, a lower detection rate of faulty modules means many faulty modules would be classified as fault-free. Once released, the faults could lead to serious field failures. The cost implications associated with the inability to detect most failure-prone modules prior to deployment could be enormous. Therefore, when evaluating models for software quality prediction, we cannot (and should not) rely on the overall accuracy (or overall error rate, computed as *1- overall_accuracy*) only.

2.2 Sensitivity, Specificity and Precision

If we consider software modules with faults as positive instances, *sensitivity* is defined as the probability that a module which contains a fault is correctly classified. In the context of software quality prediction, *sensitivity* is also called the *probability of detection* (*PD*) (Menzies et al. 2003). As discussed above, modeling techniques that produce very low *PD* are not good candidates for the software fault prediction.

The *specificity* is defined as the proportion of correctly identified fault-free modules. The *probability of false alarm (PF)* is the proportion of fault-free modules that are classified erroneously. Obviously, *PF=1-specificity*. In case of *low specificity*, many fault-free modules would be "tagged" for rigorous software verification and validation, thus unnecessarily increasing project cost and the time needed for completion. Obviously, there is always a tradeoff between *sensitivity* and *specificity*.

The *precision* index measures the chance of correctly predicting faulty modules among the modules classified as fault-prone. Either a smaller number of correctly predicted faulty

Table 2 Performance results from four projects

	PC1		*KC1*		*KC2*		*JM1*	
Method	*Acc* (%)	*PD* (%)	*Acc* (%)	*PD* (%)	*Acc* (%)	*PD* (%)	*Acc* (%)	*PD* (%)
Naïve Bayes	89.2	29.9	82.4	37.7	83.6	39.8	80.4	20.1
Logistic	92.4	6.5	85.7	20.6	82.0	38.9	81.4	11.6
IB1	91.8	44.2	83.6	40.8	78.6	50.9	76.1	36.8
J48	93.3	23.4	84.5	33.1	82.4	54.6	79.5	23.2
Bagging	93.8	16.9	85.2	24.8	83.6	47.2	81.0	19.7

modules or a larger number of erroneously tagged fault-free modules would result in a low precision. A tradeoff also exists between *sensitivity* and *precision* (Zhang and Zhang 2007). Take project *PC1*, for example. Less than *7%* of the modules contain fault(s). We ran the random forest, a tree ensemble classification method on *PC1*. The random forest algorithm builds an ensemble of classification trees from randomly selected bootstrap samples (Breiman 2001). These trees then vote to classify a data instance. Using a simple majority voting scheme, the random forest algorithm produces overall accuracy of *93.6%* and *specificity* of *98.5%*, but *sensitivity (PD)* is only *27.3%* (see Fig. 1(a)). Only *15* out of *1,032* fault-free modules were misclassified. With *21* fault-prone modules correctly classified, the *precision* is calculated to be *58.3%*. By assigning a higher threshold in random forest to the majority class, we obtained the confusion matrix in Fig. 1(b). A much higher *PD* was achieved (*74%*). Now, *57* out of *77* fault-prone modules are accurately predicted. Due to the trade-off between *sensitivity* and *specificity*, as well as the tradeoff between *sensitivity* (*PD*) and *precision*, a higher *PD* is followed by a lower *specificity* (*82.9%*) and *176* fault-free modules were incorrectly classified. The *precision* in *(b)* is only *24.5%*. But lower *precision* does not necessarily imply worse classification performance.

Each of these three measures, independently, tells us a one-sided story. For example, *sensitivity* gives us the classification accuracy for faulty modules; *specificity* provides the correct classification rate of fault-free modules. These metrics should not be used to assess the performance of predictive models unitarily. On the other hand, using several numerical performance indices to compare different predictive models does not facilitate simple algorithm comparison and/or model selection. It is difficult to observe the superiority of one model over the other since one model might have a higher *sensitivity*, but lower *precision*. We need performance metrics that provide more comprehensive evaluation of predictive software quality models. Candidate performance metrics are described in the next subsection.

2.3 G-mean, F-measure and J-coefficient

Compared to overall accuracy, geometric mean *(G-mean)* (Kubat et al. 1998), *F-measure* (Lewis and Gale 1994), and J-coefficient *(J_coeff)* (Youden 1950) tell a more honest story about a model's performance. *G-mean* indices are defined in expressions (1) and (2) below. $G\text{-}mean_1$ is the square root of the product of *sensitivity (PD)* and *precision*. $G\text{-}mean_2$ is the square root of the product of *PD* and *specificity*. In software quality prediction, it may be

Fig. 1 Confusion matrix of fault prediction based on project *PC1*. **a** Random forest with majority voting; **b** Random forest with a modified voting cutoff

Defect Predicted?

Defect Modules?	Yes	No
Yes	21	56
No	15	1,017

Accuracy = 93.6%
PD = 27.3%
1-PF = 98.5%
Precision = 58.3%

(a)

Defect Predicted?

Defect Modules?	Yes	No
Yes	57	20
No	176	856

Accuracy = 82.3%
PD = 74.0%
1-PF = 82.9%
Precision = 24.5%

(b)

critical to identify as many fault-prone modules as possible (that is, a high *sensitivity* or *PD*). If two methodologies produce the same or similar *PD*, we prefer the one with higher *specificity*. So, $G\text{-}mean_2$ is the geometric mean of two accuracies: one for the majority class and another for the minority class. *Precision* tells us among all the predicted faulty modules, how many are actually faulty.

$$G - mean_1 = \sqrt{PD \times Precision} \tag{1}$$

$$G - mean_2 = \sqrt{PD \times Specificity} \tag{2}$$

Expression (3) defines the *F-measure*. Both $G\text{-}mean_1$ and *F-measure* integrate *PD* and *precision* in a performance index. *F-measure* offers more flexibility by including a weight factor β which allows us to manipulate the weight (cost) assigned to *PD* and *precision*. Parameter β may assume any non-negative value. If we set β to *1*, an equal weight is given to *PD* and *precision*. The weight of *PD* increases as the value of β increases.

$$F - measure = \frac{\left(\beta^2 + 1\right) \times Precision \times PD}{\beta^2 \times Precision + PD} \tag{3}$$

Youden proposed the J-coefficient (*J_coeff*) to evaluate models in medical sciences (Youden 1950). El-Emam *et al.* were the first to use the J-coefficient to compare classification performance in software engineering (El-Emam et al. 2001). Expression (4) describes J-coefficient:

$$\begin{aligned} J\_coeff &= sensitivity + specificity - 1 = PD - 1 + specificity \\ &= PD - (1 - specificity) = PD - PF. \end{aligned} \tag{4}$$

When *J_coeff* is *0*, the probability of detecting a faulty module is equal to the false alarm rate. Such a classifier is not very useful. When *J_coeff*>0, *PD* is greater than *PF*, a desirable classification result. Hence, *J_coeff=1* represents the perfect classification, while *J_coeff*$=-1$ is the worst case.

These three performance indices are more appropriate in the software engineering studies than the indices which describe overall accuracy. A good software quality model should be able to identify as many fault-prone modules as possible, therefore, a high *PD* is desirable. However, if a project has a low verification budget, analysts would be interested in identifying a small number of fault-prone modules with a low false alarm rate (thus avoiding wasting time/budget by analyzing fault-free modules). In these situations, the flexibility of the G-mean and F-measure becomes important. Both *G-mean* measures, the *F-measure* and the *J_coeff* include *PD* in performance evaluation. In this way, they overcome the disadvantages of the overall accuracy measure. A low accuracy in the minority class results in a low *G-mean*, *F-measure, or J_coeff* even though the overall accuracy may be high. If all the faulty modules are predicted incorrectly, then the *G-mean* indices and the *F-measure* are zero, while the *J_coeff* may be either zero or negative. These metrics are preferable in evaluating software fault prediction models due to the fact that the cost associated with misclassifying a fault-prone module is possibly much higher than that of misclassifying a non-fault-prone module, which implies some waste of resources in verification activities.

Table 3 shows the performance results measured by the *G-mean* indices, two different *F-measures* ($\beta=1$ and $\beta=2$), and the *J_coeff* for project *PC1*. The results reflect the

Table 3 Performance results for project *PC1* based on random forests at different voting cutoffs

	Figure 1a	Figure 1b
G-mean$_1$	0.399	0.426
G-mean$_2$	0.519	0.783
F-measure (β=1)	0.372	0.368
F-measure (β=2)	0.305	0.527
J_coeff	0.258	0.569

application of the random forest classification with the same voting cutoffs as in Fig. 1. As a reminder, because of the trade-off between *PD* and *specificity* and the significantly skewed class distribution, the classifier in Fig. 1b has higher *PD*, while *precision*, *specificity* and overall accuracy are all lower.

Table 3 indicates challenges related to model evaluation with these three indices. The *G-mean$_1$* and the *F-measure (β=1)* indicate that the two voting thresholds produce very similar classification results, which is misleading. However, from a software engineer's point of view, model *(b)* is likely to be preferable to model *(a)*. This becomes obvious when we take a look at the *G-mean$_2$*, the *J_coeff or* when we assign a higher weight to *PD* in the *F-measure (β=2)*. So, without referring to *sensitivity*, *specificity* and *precision*, blindly computing and comparing some of the *F-measure* or *G-mean* could also result in a biased decision. Another difficulty is associated with the selection of parameter β in the *F-measure*. In practice, determining meaningful relative weights of precision and sensitivity is not trivial. While it is not unusual that fault prediction models are evaluated with several *F-measures*, this returns us to the problem of multiple performance indices, which we tried to avoid in the first place.

3 Graphical Evaluation Methods

In this section, we describe *Receiver Operating Characteristic (ROC)* curve, *Precision and Recall (PR)* curve, Cost Curve (Drummond and Holte 2006), and Lift Chart. To the best of our knowledge, the consideration of cost curves is novel in the software engineering literature. These graphs are closely related, being derived from the confusion matrix. Ling and Li (1998) mentions a close relationship between *ROC* and lift chart without offering a mathematical justification. In (Vuk and Curk 2006), Vuk and Curk define a common mathematical framework between *ROC* and lift chart, and the Area Under the ROC Curve (*AUC*) and the Area Under the Lift Chart. In spite of similarities, each curve reveals different aspects of classification performance, making them worth considering in software engineering projects.

3.1 ROC Curve

Many classification algorithms allow users to define and adjust a threshold parameter in order to generate appropriate models. When predicting software quality, a higher *PD* can be produced at the cost of higher *PF* and vice versa. The (*PF,PD*) pairs generated by adjusting the algorithm's threshold form an *ROC* curve. ROC analysis is a more general way to measure a classifier's performance than numerical indices (Yousef et al. 2004). An ROC curve offers a visual of the tradeoff between the classifier's ability to correctly detect fault-prone modules (*PD*) and the number of incorrectly classified fault-free modules (*PF*).

The *Area Under the ROC curve* (denoted *AUC*) is a numeric performance evaluation measure directly associated with an ROC curve. It is very common to use *AUC* to compare the performance of different classification methods based on the same data. However, the entire region under the curve may not be of interest to software engineers. For instance, the regions associated with very low probability of detection (region C in Fig. 2a) or very high probability of false alarm (region B) or both (region D), typically indicate poor performance. With a few possible exceptions (for example safety critical systems, in which risk aversion drives the development process), only the performance points associated with acceptable *PD* and *PF* rates (region A) are likely to have a practical value for software engineers.

Therefore, in software engineering studies, *the area under the curve within region A* (denoted *AUCa*) is typically a more meaningful method to compare models than the standard *AUC*. Fig. 2b shows the ROC curves of classifiers *nb* (Naïve Bayes) and *j48* on KC4 project. Using the trapezoid rule, we calculated *AUCs* to be *0.750* and *0.745*, respectively. From AUC point of view, *nb* would be a preferred classifier, but after we calculate *AUCa measures*, we find that their values are *0.272* for *nb* and *0.368* for *j48*. The *AUCa* of *j48* is greater than that of *nb*, indicating a preference for the *j48* model, the outcome consistent with the impression a software engineer would gather from looking at the *ROC* curves in Fig. 2b. This example illustrates that comparing quality prediction models based on their *ROC* curves and the associated standard *AUC* measures is not as straightforward as one would expect.

Another simple non-parametric method that utilizes *AUC* can serve the comparison in software engineering studies (Braga et al. 2006). The main idea is to compare the *ROC* curves using a collection of sampling lines from a reference point; these lines sample the *ROC* space. The intersection points of these lines with the *ROC* curves can be identified and their Euclidean distances to the reference point computed. This method allows the identification of performance points in which one classifier is better than the other. For example, referring to Fig. 2a, the reference point can be the coordinate (*PF, PD*)=(*0.50, 0.50*). Starting from this point, *N* lines with different slopes can be drawn randomly into the

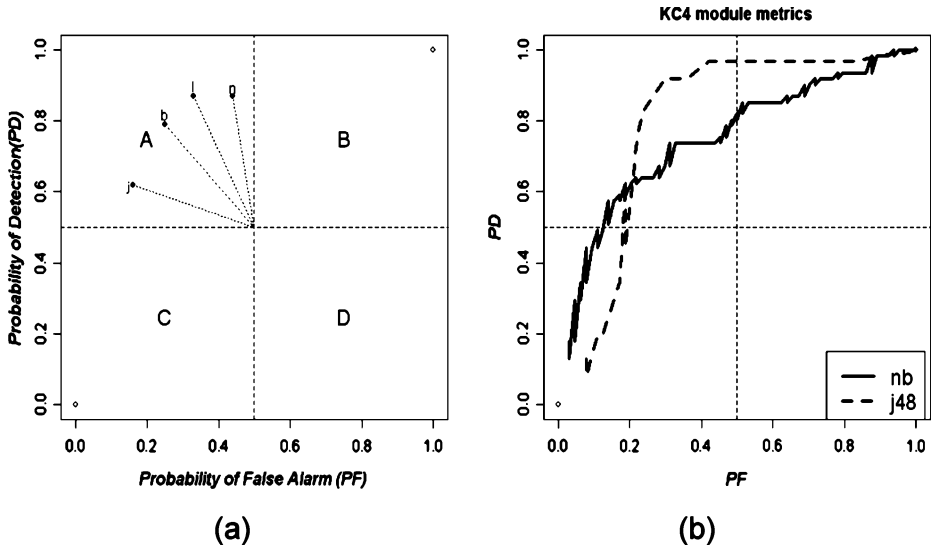


Fig. 2 ROC analysis. **a** Four regions represent performance parameters of software fault prediction models. **b** ROC curves from two classifiers Naïve Bayes (nb) and j48 on project KC4

Region A. Suppose we are evaluating two *ROC* curves representing different classifiers. The distances from point (*0.50, 0.50*) to the *N* intersection points on each *ROC* can be computed. Longer distances along sampling lines would indicate better classification within region A of the ROC space.

The utility of this comparison method extends to classifiers which lack the threshold parameter. Such algorithms are only capable of producing a single point in the ROC space, a (*PF, PD*) pair. Rather than measuring distances from an arbitrary reference point, one can calculate the distance from the perfect classification point (0, 1) to the (*PF, PD*) pair point (Menzies et al. 2007):

$$\text{Distance from Perfect Classification} = \sqrt{\theta \times (1 - PD)^2 + (1 - \theta) \times PF^2}, \qquad (5)$$

where θ is a parameter ranging from 0 to 1, used to control weights assigned to *1-sensitivity (i.e., 1-PD)* and *1-specificity (i.e., PF)*. The smaller the distance, i.e., the closer the point is to the perfect classification, the better the performance of the associated classifier. Figure 3 plots four (*PF, PD*) pair points in a 2-dimentional space. Points *A* and *B* represent the classification performance of the random forest classifier on project PC1 from Fig. 1a and b, respectively. C and D are not results of any specific algorithm and are used to demonstrate performance comparison. The (*PF, PD*) pairs for *C* and *D* are (0.10, 0.80) and (0.20, 0.90), respectively. If we set θ to be 0.5, then *C* and *D* have an equal distance from (0, 1). From a software engineer's point of view, are they really equivalent? The answer is dependent on the rationale behind the value of θ. If misclassifying a faulty module results in consequences that outweigh falsely "tagging" a fault-free module, then more weight should be given to *1-PD*. In this case, algorithm *D* would provide a better classification performance than *C*.

The distance from the perfect classification point (*0,1*) is appropriate to choose the best point from a convex hull of an *ROC* curve too. This method can assist software engineers in determining the "best" threshold for a classifier, given a project dataset and an appropriate value of θ.

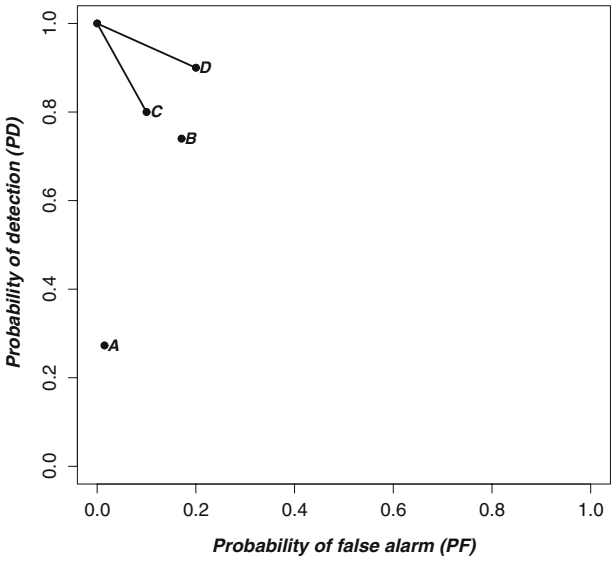


Fig. 3 Points in ROC space represent different classification performance

3.2 Precision-recall Curve

Precision-Recall (PR) curve presents an alternate approach to the visual comparison of classifiers (Davis and Goadrich 2006). *PR* curve can reveal the difference between algorithms which is not apparent from an *ROC* curve. In a PR curve, x-axis represents *recall* and y-axis is *precision*. *Recall* is yet another term for *PD*.

Figure 4 shows an example. Looking at the ROC curves for project PC5, it is difficult to tell the difference among the three classifiers: Naive Bayes (nb), Random Forest (RF), and IBk. However, their PR curves allow us to understand the difference in their performance: Random Forest algorithm performs better than Naïve Bayes and Naïve Bayes has an advantage over IBk.

In ROC curves, the best performance indicates high *PD* and low *PF* in the upper left-hand corner. PR curves favor classifiers which offer high *PD* and high *Precision, i.e.*, the ideal performance is in the upper right-hand corner. In Fig. 4a, the performance of the three classifiers appears to approach close to the optimal left-hand upper corner. However, the PR curve of Fig. 4b indicates that there is still plenty of room for the improvement of classification performance. We do not contend that *PR* curves are better than *ROC* curves. We only recommend that when *ROC* curves fail to reveal differences in the performance of different classification algorithms, *PR* curves may provide adequate distinction.

3.3 Cost Curve

The aforementioned graphical representations do not consider the implicit cost of misclassification. In fact, the basic assumption is that the cost to misclassify a fault-prone module as fault-free is the same as the cost of misclassifying a fault-free module as a fault-prone. Adams and Hand (1999) defined loss difference plots to take the advantage of the misclassification cost ratio. Loss difference plot is considered the predecessor of cost curves which we introduce below.

Proposed by Drummond and Holte (2006), cost curve is a visual tool that allows us to describe classifier's performance based on the cost of misclassification. Its *y-axis* represents normalized expected misclassification cost. It indicates the difference between the maximum and the minimum cost of misclassifying faulty modules. The *x-axis* represents the probability cost function, denoted *PC(+)*.

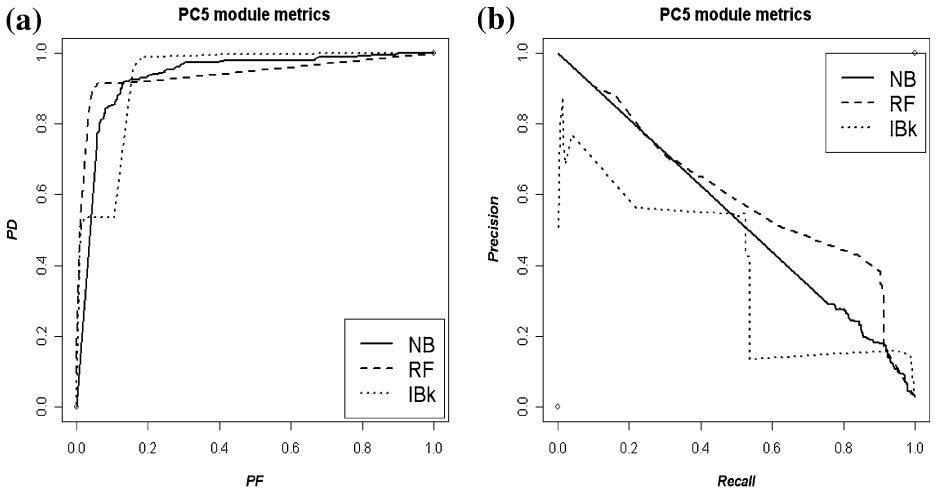


Fig. 4 **a** ROC curve and **b** PR curve of models built from PC5 module metrics

Let us denote the faulty module with a "+" and a fault-free module with a "–". *C(+|−)* denotes the cost of incorrectly predicting a fault-free module as faulty. *C(−|+)* represents the cost of misclassifying a faulty module as being fault-free. *p(+)* and $p(-)$ are the probabilities of a software module being faulty or fault free, $p(+) + p(-) = 1$. These probabilities become known only after the deployment of the model, since the proportion of faulty modules in model's training and test sets are the approximations of the proportion of faulty modules the model will encounter during its field use. Cost curves support visualization of model's performance across all possible values of $p(+)$ and $p(-)$, offering performance predictions for various deployment environments.

Equations 6 and 7 present the formulae for computing the values that appear on *x-axis* and *y-axis*:

$$x - axis = PC(+) = \frac{p(+) \times C(-|+)}{p(+) \times C(-|+) + p(-) \times C(+|-)}, \tag{6}$$

$$\text{y} - \text{axis} = \text{NormalizedExpectedCost} = (1 - PD - PF) \times PC(+) + PF, \tag{7}$$

where PF is the probability of false alarm and PD is the probability of detection, both defined in Section 2.2. Let us define the *misclassification cost ratio* $\mu = C(+|-)$: $C(-|+)$. Equation 6 can be rewritten as:

$$x - axis = PC(+) = \frac{1}{1 + \frac{1-p(+)}{p(+)}\mu} \tag{8}$$

From Eq. 8, we observe that the x-axis is determined by only two parameters, *p(+)* and μ.

In cases when the proportion of faulty modules, *p(+)*, is known a cost curve visualizes models which cover a range of misclassification cost ratios. Knowing the values of *p(+)* for the datasets used in this study make the illustration of such use easy. In stable development environments, the proportion of faulty modules can be estimated from the past performance. When *p(+)* is unknown, cost curve is typically used to evaluate model performance for a set of assumed misclassification ratios μ over a range of expected proportions of faulty modules in the software project. This is a common case in practice. While inexact, the estimation of the misclassification ratio μ should be inferred from project's characteristics such as severity, priority, failure costs, etc. In both cases, cost curve can provide guidance for the selection of fault prediction models relevant for the specific project's environment.

A sample cost curve diagram is shown in Fig. 5. The diagonal line connecting *(0,0)* to *(1,1)* stands for a trivial classifier that always classifies all the modules as fault-free. The other diagonal line, connecting *(0,1)* to *(1,0)*, stands for another trivial classifier that always classifies all the modules as fault-prone. The horizontal line connecting *(0,1)* to *(1,1)* represents the extreme situation when the model misclassifies all the modules. The *x-axis* represents the ideal model which correctly classifies all the modules.

An ROC curve connects a set of (PF, PD) pairs. Cost curves are generated by drawing a straight line connecting points (0, PF) and (1, 1-PD), corresponding to a (PF, PD) point in the ROC curve. After drawing all the straight lines that have the counterpart points in the ROC curve, the lower envelope of the cost curve is formed by connecting all the intersection points from left to right. Figure 6 shows an example of a cost curve resulting from the application of *Logistic* classification model to project KC4. The lower envelope of the cost curve corresponds to the convex hull in the ROC curve. ROC curves and cost curves are closely related: a point in the ROC curve corresponds to a line in the cost curve.

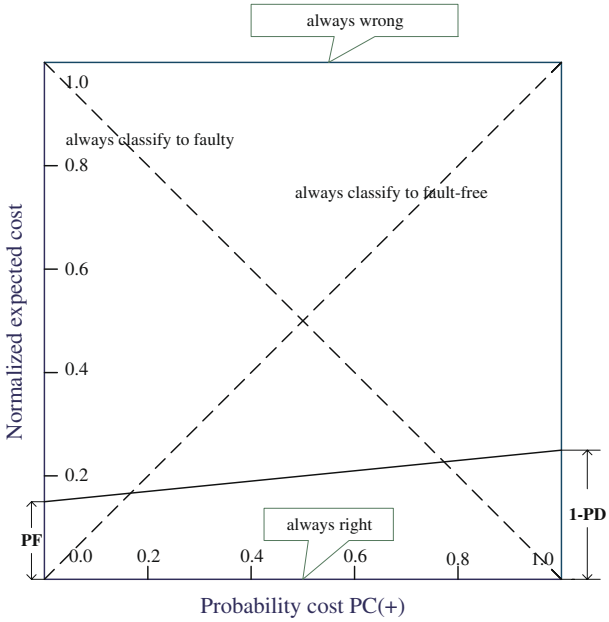


Fig. 5 Typical regions in cost curve

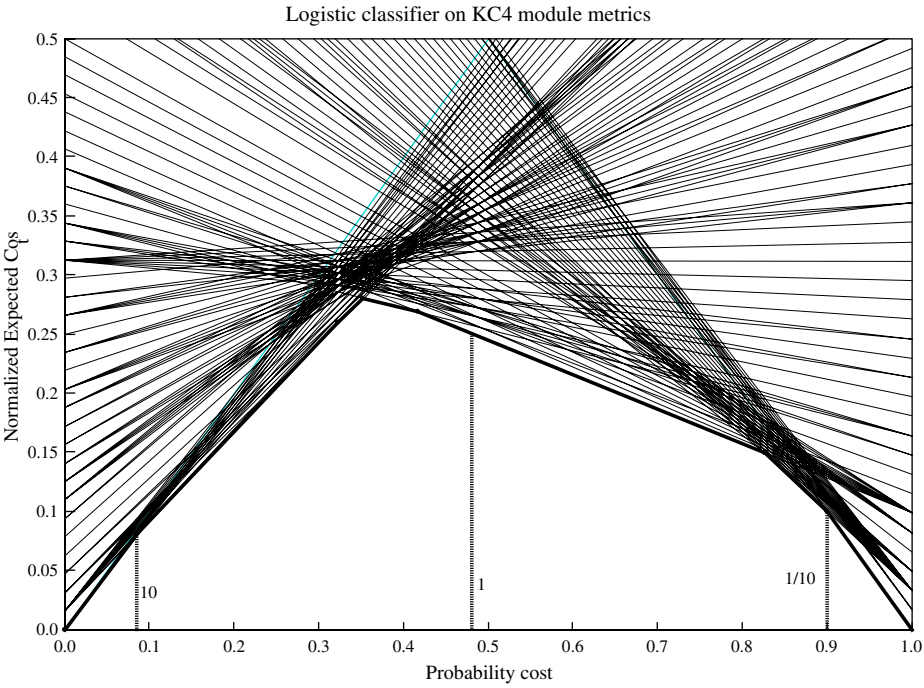


Fig. 6 The cost curve of logistic classifier using KC4 project data

If the costs of misclassifying faulty modules and misclassifying fault-free modules are the same (admittedly a rare case in any software engineering project), $C(+|-) = C(-|+)$, the corresponding point in the probability cost function *PC(+)=0.48* (see Table 1, the proportion of faulty modules, *p(+)*, in KC4 is 48%). In Fig. 6 this point is denoted by the vertical line at $x = PC(+) = 0.48$. Therefore, if the misclassification cost ratio is *1*, we should choose the classifier that offers the minimal expected cost along this vertical line. When *PC(+)<0.48*, the misclassification cost of fault-free modules is greater than that of misclassifying a fault-prone module. The vertical line at $x = PC(+) = 0.0845$ indicates the place where the cost ratio *C(+|−)* to *C(−|+)* is *10*. The cost region where *PC(+)>0.48* represents models which are adequate for "risk adverse" projects, those where misclassifying a fault-prone module is significantly more consequential. The vertical line at *x=PC(+)=0.90* indicates the situation where $C(+|-)$: $C(-|) = 1 : 10$. Cost curves open significant opportunities for cost-based software quality management. Misclassification costs can now dictate the preference for the model and model parameters which are the most appropriate for the given project.

While the goal of *ROC* and *PR* curve analysis is to maximize the area under the curve, the goal of cost curve is to minimize the misclassification cost, that is, minimize the lower envelope area. The smaller the area under the lower envelope boundary is, the better the performance of the classifier and, consequently, the better the expected software quality cost–benefit ratio. However, it is likely that a single model will not be equally good in all cost regions. If at some point in time a project evolves to an understanding that the misclassification cost differential is not the same one used in the past, the project may decide to change the quality model it uses even though the underlying metric distributions in the dataset remain the same. Cost curves support this type of decisions.

Cost curve reveals the differences among classifiers from perspectives that are not obvious in *ROC* and *PR* curves. Figure 7 demonstrates this using KC4 dataset as an example. The cost curve, shown in panel (c) of Fig. 7, provides a clear answer to the comparison of two classifiers: Logistic classifier is better than Naïve Bayes (*nb*) because its lower envelope is always below that of Naïve Bayes. Deriving this conclusion from *ROC* or *PR* curves would be difficult.

It is not our intension to advocate that cost curve is always the best method for comparing the performance of fault prediction models. *ROC*, *PR* and cost curves describe classifier performance from different perspectives and they are all useful. When comparing several models, it is a good practice to plot *ROC* curves, *PR* curves, and cost curves because they provide complementary information for model selection.

3.4 Lift Chart

In practice, every software project has a finite time and budget constraints for verification and validation activities. Given the fault prediction model, the question that typically arises is how to utilize available resources to achieve the most effective quality improvement. Lift chart (Witten and Frank 2005), known in software engineering as Alberg diagram (Khoshgoftaar et al. 2007; Ostrand et al. 2005; Ohlsson and Alberg 1996; Ohlsson et al. 1997), is another visual aid for evaluating classification performance. Lift is a measurement of the effectiveness of a classifier to detect fault-prone modules. It calculates the ratio of correctly identified faulty modules with and without the predictive model. Lift chart is especially useful in situations when the project has limited resources to apply verification activities to, say, *5%* of the modules. Which model maximizes the probability that the selected *5%* of the project's modules contain faults? This type of prediction is common in the literature. Ostrand, Weyuker and Bell focus on predicting and ordering top 20% of files which contain the most faults (Ostrand et al. 2005). Arisholm and Briand predict fault-prone components

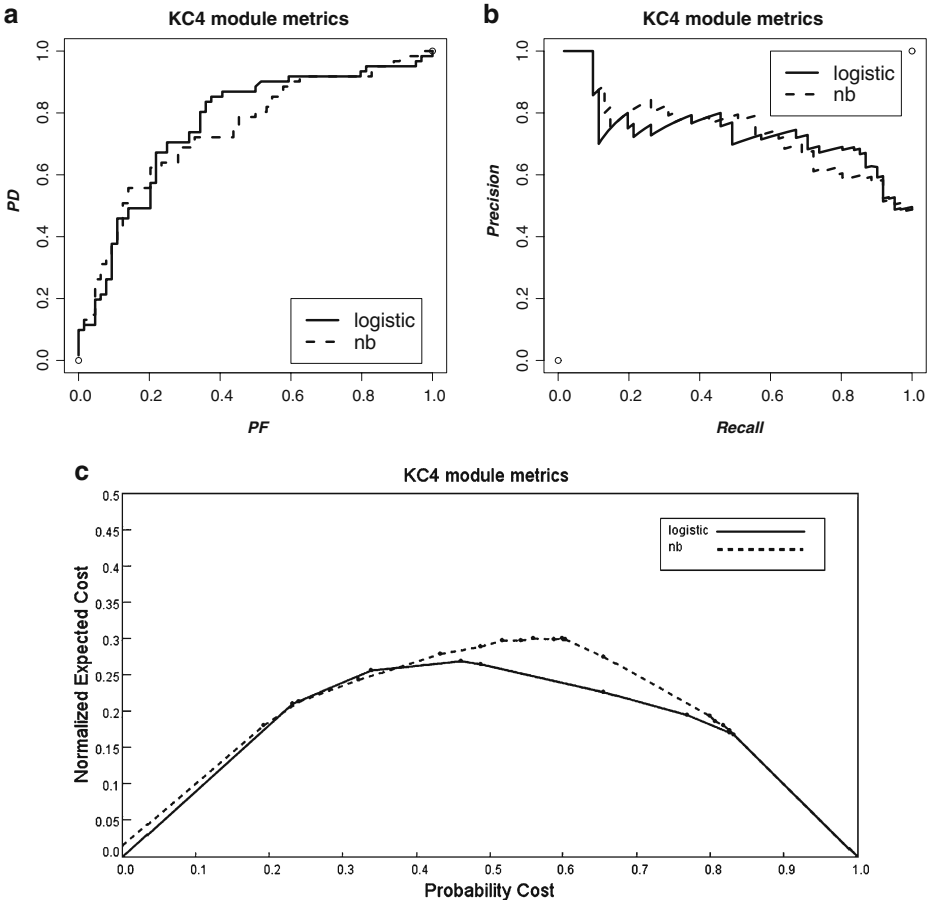


Fig. 7 **a** ROC; **b** PR curve; and **c** Cost curve represent two fault prediction models using KC4 project data

in a java legacy system and argue that a prediction model is not practical if the predicted *X%* of faults requires inspection of more than *X%* of the code (Arisholm and Briand 2006).

Lift chart analysis starts by ranking all the modules with respect to their chance of containing fault(s). The ranking methods can vary (Khoshgoftaar et al. 2007; Kubat et al. 1998; Ostrand et al. 2005; Ohlsson and Alberg 1996; Ohlsson et al. 1997; Witten and Frank 2005). For example, multiple linear regression models calculate the expected number of faults in a module, Naive Bayes models output a score indicating the likelihood that the module belongs to a faulty class and an ensemble algorithm, such as random forest, counts the voting score. Once the modules are ranked, we calculate the number of faulty modules in the specific rank (from 0 to 100%). In the lift chart, the *x-axis* represents the percentage of the modules considered, and the *y-axis* indicates the corresponding detection rate within this sample. The lift chart consists of a baseline and a lift curve. The lift curve shows the detection probability resulting from the use of the predictive model, while the baseline indicates the proportion of faulty modules in the dataset. The greater the area between the lift curve and the baseline is, the better the performance of the classifier.

Figure 8a depicts lift charts of two classifiers, J48 and logistic, applied to project PC5. PC5 contains *17,186* modules and *3%* of them are faulty. The baseline indicates the proportion of

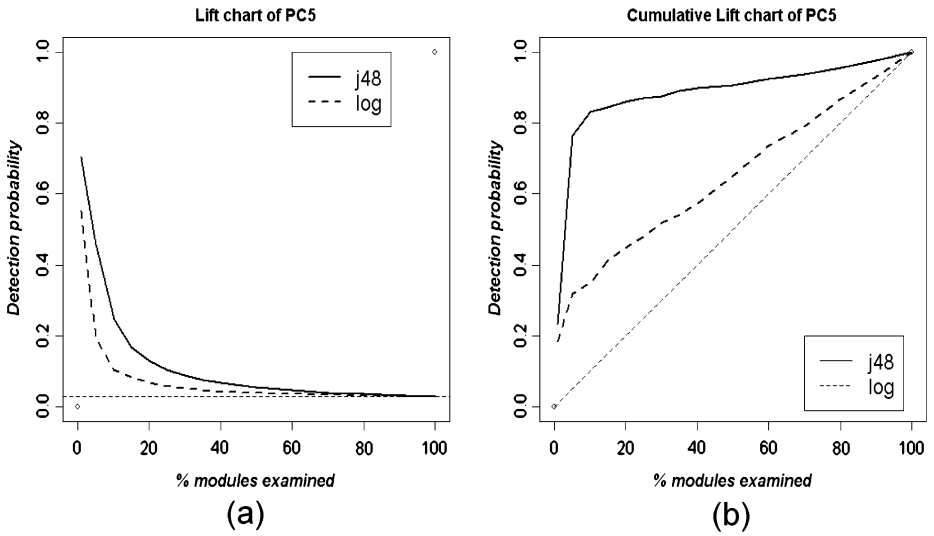


Fig. 8 Lift chart and cumulative lift chart of PC4 module metrics

faulty modules (*3%*). Say budget constraints allow us to analyze *5%* of the PC5 modules (*0.05× 17,186=859* modules). Without a predictive model, we would have a *3%* chance to randomly select faulty modules, i.e., out of *859* randomly selected modules only *26* modules would be expected to contain faults. Using *J48*, there is *45.85%* chance to capture faulty modules within the top 5%. Consequently, this model is expected to expose 394 (*45.85%× 859=394*) faulty modules to verification analysis, with a *lift factor* of more than *15*.

Figure 8b demonstrates another variation of lift charts—a cumulative lift chart. The *x-axis* is the same as above, but the *y-axis* stands for the cumulative proportion of detected faulty modules. The greater the area under the curve is, the better the classifier's performance.

4 Statistical Comparisons of Classification Models

Drawing sound decisions from performance comparison of different classification algorithms is not simple. Statistical inference is often required (Conover 1999; Siegel 1956). The purpose of performance comparison is to select the best model(s) out of several candidates. Suppose there are *k* models to compare. The statistical hypothesis is:

H_o: *There is no difference in the performance among k classifiers.*

vs.

H_1: *At least two classifiers have significantly different performance.*

When more than two classifiers are under comparison, a multiple test procedure may be appropriate. If the null hypothesis of equivalent performance among *k* classifiers is rejected, we can proceed with a post-hoc test.

Demsar (2006) overviewed theoretical work on statistical tests for classifier comparison. When the comparison includes more than two classifiers over multiple datasets, he recommends the Friedman test followed by the corresponding post-hoc Nemenyi test. The Friedman test and the Nemenyi test are nonparametric counterparts for analysis of variance (ANOVA) and Tukey

test parametric methods, respectively. Demsar advocates these tests largely due to the fact that nonparametric procedures make less stringent demands on the data. However, nonparametric tests do not utilize all the information available, as the actual data values (typically numerical performance indices such as *AUC*) are not used in the test procedure. Instead, the signs or ranks of the observations are used. Therefore, parametric procedures will be more powerful then their nonparametric counterparts, when justifiably used.

To illustrate the application of nonparametric tests in software engineering studies, we follow nonparametric test procedure recommended by Demsar (2006). The Friedman test and the Nemenyi test are implemented in the statistical package R (http://www.r-project.org/). In our experiments, the measure of interest is the mean *AUC* estimated over 10 by 10 cross validation, using *95%* confidence interval ($p=0.05$) as a threshold to judge the significance ($p<0.05$). Of course, any numerical performance index discussed in this paper can form the basis for statistical analysis. Below, the Friedman test tests whether there is a difference in the performance among *6* classification algorithms over *8* datasets (all listed in Section 1.1). Provided that the Friedman test indicates statistically significant difference exists, we will need to compare classifiers to determine which classifier performs better over these *8* data sets using the Nemenyi test.

We implement the Friedman test as follows (Demsar 2006):

$$F_f = \frac{(N-1)x_f^2}{N(k-1)-x_f^2}, \qquad \text{where} \qquad x_f^2 = \frac{12N}{k(k+1)}\left[\sum_j R_j^2 - \frac{k(k+1)^2}{4}\right], \tag{9}$$

k is the number of classifiers, N is the number of data sets, R_j is the average rank of a classifier j over multiple data sets. $R_j = \frac{1}{N}\sum_j r_i^j$, where r_i^j is the rank of the *jth* classifier on the *ith* data. F_f follows the F-distribution with $k-1$ and $(k-1)(N-1)$ degrees of freedom and the critical values available from any statistical textbook.

The Nemenyi test is a post-hoc test of the Friedman test, applied if the null hypothesis is rejected. It will compare all classifiers with each other. The critical difference (*CD*) of the Nemenyi test is calculated as $CD = q_a\sqrt{\frac{k(k+1)}{6N}}$, where q_a is the critical value of the Nemenyi test (Demsar 2006). The performance difference between the two classifiers is significant if the difference of average ranks between them is larger than the value of *CD*. In our case, we have $N=8$ data sets and $k=6$ classifiers, The average ranks of *IB1*, *J48*, *NB*, *logistic*, *bagging*, and *random forest* are 5.50, 5.25, 3.625, 3.125, 2.5, and 1, respectively. The Friedman test checks the null hypothesis:

$$x_f^2 = \frac{12\times 8}{6(6+1)}\left[5.50^2 + 5.25^2 + 3.625^2 + 3.125^2 + 2.5^2 + 1^2 - \frac{6(6+1)^2}{4}\right] = 33.07,$$

$$F_f = \frac{(8-1)\times 33.07}{8(6-1)-33.07} = 33.41.$$

With $k-1=5$ and $(k-1)(N-1) = 5\times 7 = 35$ degrees of freedom, the critical value of F-distribution is *2.48*. Because 33.41>2.48, the null hypothesis which states that there is no difference in the performance of these six models over the eight data sets is rejected. Next, the Nemenyi post-hoc test compares the performance of classifiers. With six classifiers, the critical value q_a is 2.85 (Demsar 2006). Hence, the critical difference is $CD = 2.85\sqrt{\frac{6(6+1)}{6*8}} = 2.67$

Figure 9 shows the outcome of the Nemenyi post-hoc tests. The numbers in the scale represent the average rank; the higher the rank, the worse the performance of a classifier. Therefore, from the worst towards the best, the order of models is IB1, J48, Naïve Bayes,

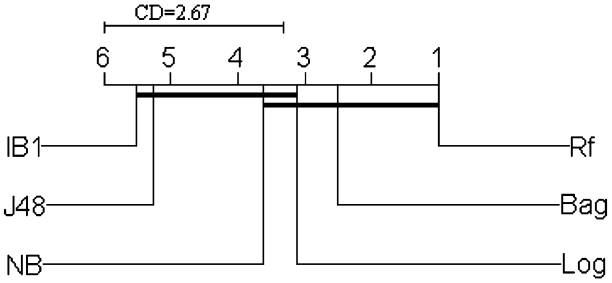


Fig. 9 Comparison of 6 classifiers over 8 datasets using the Nemenyi test with 95% CI

Logistic, Bagging, and Random Forest. When the difference between the average ranks or two models is smaller than the value of *CD*, the difference in their performance is not significant, as indicated by the bold straight lines. Figure 9 indicates that our fault prediction models form two performance clusters: IB1, J48, Naïve Bayes, and logistic models form one and Naïve Bayes, logistic, bagging, and random forest form the other.

While the example above illustrates the application of statistical testing to a numerical performance index (AUC), similar tests can be applied to the graphical tools for performance evaluation. Macskassy et. al. review how to construct confidence intervals around *ROC* curves, either as point-wise (Macskassy et al. 2005a) or global confidence bands (Macskassy et al. 2005b). In the point-wise approach, the user tests the difference between two *PD* values for a given *PF* value. Statistical analysis of differences between two AUCs is an example of global confidence bands method. The same type of statistical testing can be applied to *ROC* curves, *PR* curves and lift charts. In the rest of this section, we concentrate on the construction of confidence intervals for cost curves, the topic not addressed previously in the software engineering literature.

Classifier's performance is derived from a confusion matrix. In a cost curve, as well as in other performance analysis graphs, a confidence interval is generated from a series of confusion matrices. We illustrate the use of confidence bounds on cost curves, following the procedure outlined by Drummond and Holte (2006). The procedure is based on the resampling of typically *500* confusion matrices using the bootstrap method. The *95%* confidence interval for a specific PC(+) value, for example, is obtained by eliminating the highest and the lowest *2.5%* of the normalized expected cost values.

In some ranges of the curve, the performance of two classifiers may be significantly different, but in the other ranges it may not. Figure 10 shows an example of the difference in performance between two classifiers, IB1 and j48 on PC1 dataset. The shaded area in Fig. 10 represents the *95%* confidence interval. There are three lines inside the shaded area. The middle line represents the difference between the means of IB1 and j48. The other two lines along the edges of shaded area represent the *95%* confidence interval of the difference between the mean of two classifiers. If the confidence interval contains the (horizontal) zero line, then there is no significant difference between the two classifiers, otherwise, there is. The larger the distance of the confidence interval from the zero line, the more significant the difference between the two classifiers. Referring to Fig. 10, when PC(+) is in the range of *(0.0, 0.1)*, the confidence interval is above the zero, indicating J48 outperforms IB1. In the range *(0.27, 0.59)*, the confidence interval is below the zero line, indicating that IB1 performs better than J48. Between *(0.1, 0.27)* and *(0.59, 0.64)*, the confidence interval contains the zero line indicating no significant difference between the two classifiers. When $PC(+)>0.64$, the performance of the two classifiers is the same. Given that the proportion of faulty modules in project PC1 is *7%*, at $PC(+)=0.07$ the misclassification cost for fault-

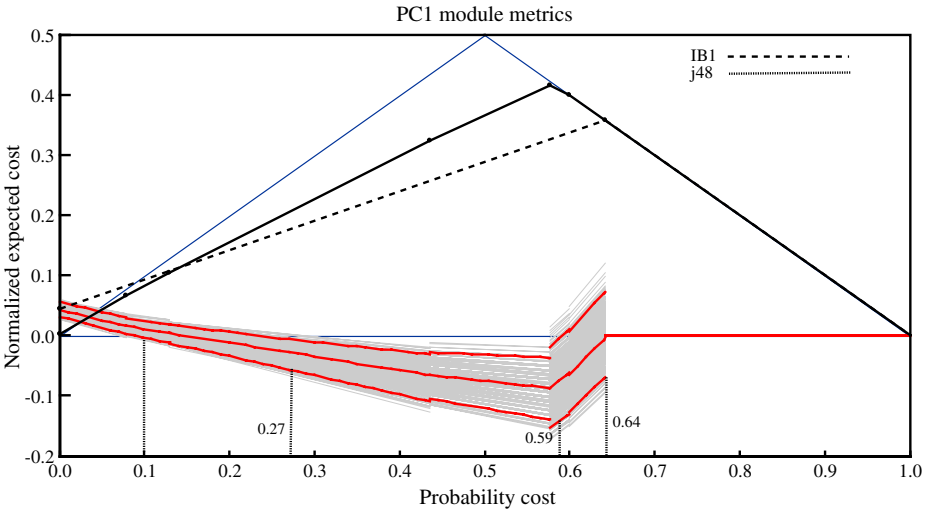


Fig. 10 The 95% cost curve confidence interval comparing IB1 and j48 on PC1

prone and fault-free classes is the same, $C(+|-) = C(-|+)$. When *PC(+)<0.07* the cost of misclassifying fault-free modules outweighs the misclassification of fault-prone modules. The opposite is true for *0.07<PC(+)*. Cost curve analysis shows that (1) j48 is a bad choice when *PC(+)<0.07* (worse even than the trivial classifier); (2) j48 outperforms IB1 in *(0.27, 0.59)* region; (3) in other regions of the cost curve, IB1 and j48 perform similarly.

The confidence band in the cost curve reveals aspects which cannot be inferred from the statistical analysis of *AUC* in *ROC* curves. When misclassification costs are known or can be guessed, cost curves and their statistical analysis provide the most meaningful guidance for model selection.

5 Experimental Evaluation

In this section, we mimic the analysis a software practitioner would perform when developing and selecting fault prediction models. We start by building the six models using classifiers listed in Section 1.1. For the purpose of analysis presented in this section, we eliminated models developed using the random forest algorithm. The random forest outperforms other classification models in many of the measures of interest, thus making the problem of model selection rather trivial. Since the purpose of this paper is the description of the model comparison and selection process, eliminating one of the classifiers does not reduce the generality of our recommendations.

5.1 PC1 Dataset

We would like to select the most appropriate classifier for project *PC1* from the MDP repository. To facilitate the comparison, in Table 4 we highlight the best performers according to each numerical index. Using the default parameters for each classifier, as offered in Weka toolkit, the superiority of *IB1* classification algorithm appears noteworthy, as it provides the largest value on majority of the performance indices. If we check *PD*, and

Table 4 Numerical performance indices for project *PC1*

Indices	Naïve Bayes	Logistic	IB1	J48	Bagging
PD	0.299	0.065	**0.442**	0.234	0.169
1−PF	0.936	0.988	0.954	0.985	**0.995**
Precision	0.259	0.289	0.415	0.540	**0.732**
Overall accuracy	0.892	0.924	0.918	0.933	**0.938**
G-mean$_1$	0.278	0.137	**0.428**	0.356	0.352
G-mean$_2$	0.529	0.253	**0.649**	0.480	0.410
F-measure (β=1)	0.278	0.106	**0.428**	0.327	0.275
F-measure (β=2)	0.290	0.077	**0.436**	0.264	0.200
ED (θ=0.5)	0.498	0.661	**0.396**	0.542	0.588
J_coeff	0.235	0.053	**0.396**	0.219	0.164

set the acceptability threshold to *0.40* then only *IB1* is up to this standard. *G-mean's*, *F-measure*, *J_coeff* or distance from the perfect classification all point to *IB1* as our candidate model. But, if we use *overall accuracy*, *precision* or *specificity*, we would choose the model developed by *bagging* classifier. However, the probability of detection (*PD)* from bagging appears poor (~17%) making its suitability for a software engineering project questionable.

However, as mentioned earlier, numerical performance indices only reveal a part of the overall evaluation story. The default operational points of classifiers listed in Table 4 offer a very particular point of view and hide the multitude of tuning options which can significantly alter the outcome of evaluation.

For this reason, analysis must continue with the visual model performance evaluation methods. Figure 11 shows the *ROC* curves corresponding to these five models. The analysis of *AUC*s orders the models as follows: *Bagging>Logistic>IB1>Naïve Bayes>J48*. The actual *AUC*s are shown in Table 5. Mean *AUC* values match the order observed from the ROC curves. However, if the curves tangle together, as they do in Fig. 11, it is difficult to assess whether the differences are statistically significant.

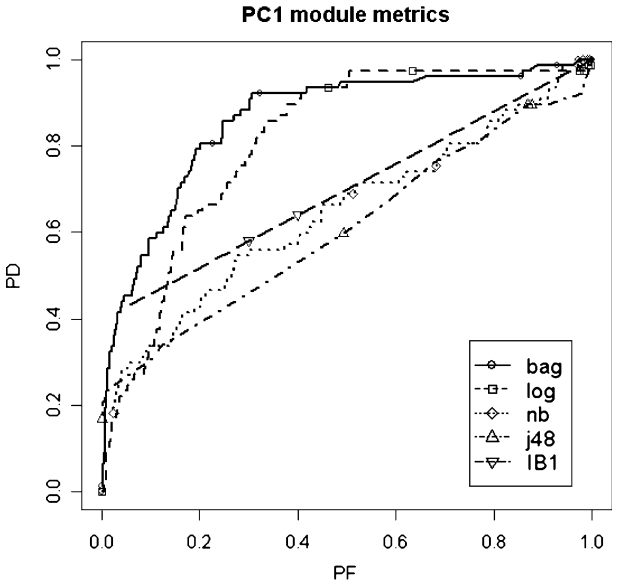


Fig. 11 ROC curves of PC1 module metrics

Table 5 AUC and AUCa of PC1 fault prediction models

Dataset	Model	AUC	AUCa
pc1	IB1	0.694	0.14
	J48	0.665	0.096
	Bag	**0.840**	**0.496**
	Log	0.814	0.428
	NaiveBayes	0.669	0.112

We next apply the Friedman test and the Nemenyi test, using the AUC values from 10 by 10 cross validation experiments. Although we now compare model performance using multiple experiments from a single data set, the validity of this statistical inference procedure is not diminished. Figure 12 shows the test outcomes from the Nemenyi test. The average rank orders the five models as follows: J48, NaiveBayes, IB1, logistic, and bagging. From Fig. 12, we can conclude that: (1) The average rank of J48 and Naïve Bayes is the highest, hence they offer the worst performance; (2) The average rank of Bagging is *1*, offering superior performance among the five models, although using 95% confidence interval its performance cannot be distinguished from models developed using Logistic classifier; (3) The difference in performance between J48, Naïve Bayes and, IB1 is not statistically significant; The performance differences between IB1 and Logistic are also not significant; (4) Other comparisons reveal statistically significant differences in model performance.

From the ROC curves, mean AUC values and the outcome of the nonparametric tests, we would likely select the fault prediction model built by the Bagging classification algorithm. Note that this is a different outcome compared to the classifier selection from the numerical indices. Let us consider this choice in the cost curve diagram shown in Fig. 13.

When the misclassification cost for fault-prone and fault-free classes in *PC1* project is the same, we have *PC(+)=0.07*. This is indicated by the vertical line in the cost curve of Fig. 13. If PC1 is a cost adverse project (meaning that the cost of misclassifying fault-prone modules as fault-free is lower than the other way around) the region of interest for performance comparison covers probability cost *PC(+)<0.07*. If PC1 is a risk adverse project (the cost of misclassifying fault-free modules as fault-prone is lower, implying the project is trying to minimize the risk of post deployment failure) the region of interest covers probability cost *PC(+)>0.07*. When *PC(+)<=0.07*, all our models have misclassification rate/cost similar or worse than a trivial classifier (which classifies all modules as fault-free). Consequently, no model is particularly useful if PC1 project falls into the cost adverse category. Using any of these *5* models can only be justified in the "risk adverse" region of the cost curve. The misclassification cost trend of these *5* classifiers indicates the

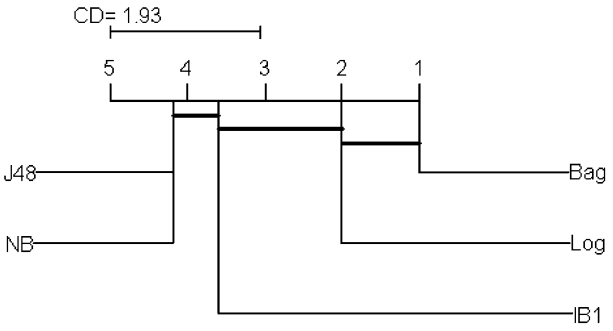


Fig. 12 Comparison of the five classifiers on PC1 data using the Nemenyi test and 95% CI

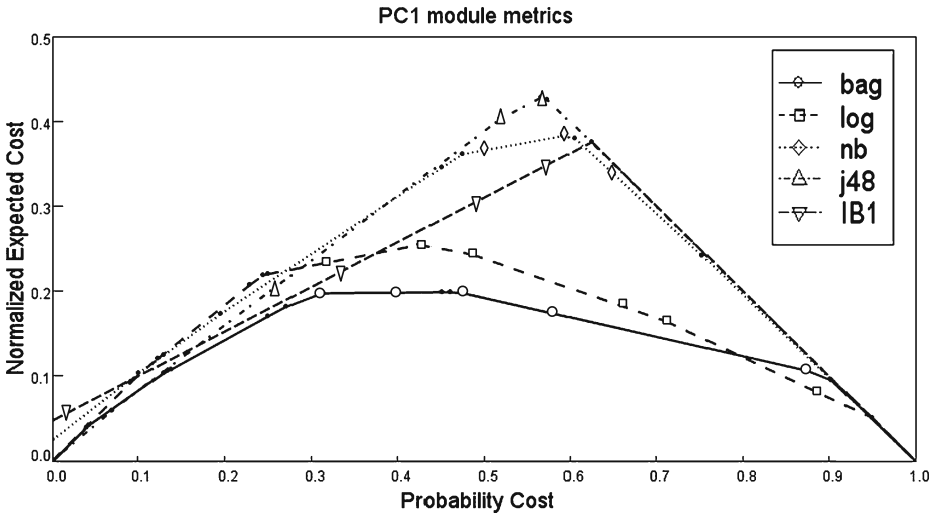


Fig. 13 Cost curves for project PC1

same tendency as ROC curves from Fig. 11: *Bagging<Logistic<IB1<NaiveBayes<J48*. Generally, *bagging* has the minimal cost among the five models and it is consistently within the envelope of the trivial classifier. Its use can contribute to software quality assurance across the entire range of probability cost. Thus, when misclassification cost is considered, the model generated by bagging algorithm is the best pick.

We conducted pairwise statistical comparisons of the cost curve models too. Figure 14 shows an example, where we compare j48 and Bagging models. When $0 =< PC(+) <= 0.31$ and $0.78 =< PC(+) <= 1$, the normalized expected cost *0* (i.e., *y=0*) is inside the *95%* confidence interval. Therefore, there is no significant difference between these two classifiers in these ranges. We observe a significant difference in the region *0.31<PC(+)<*

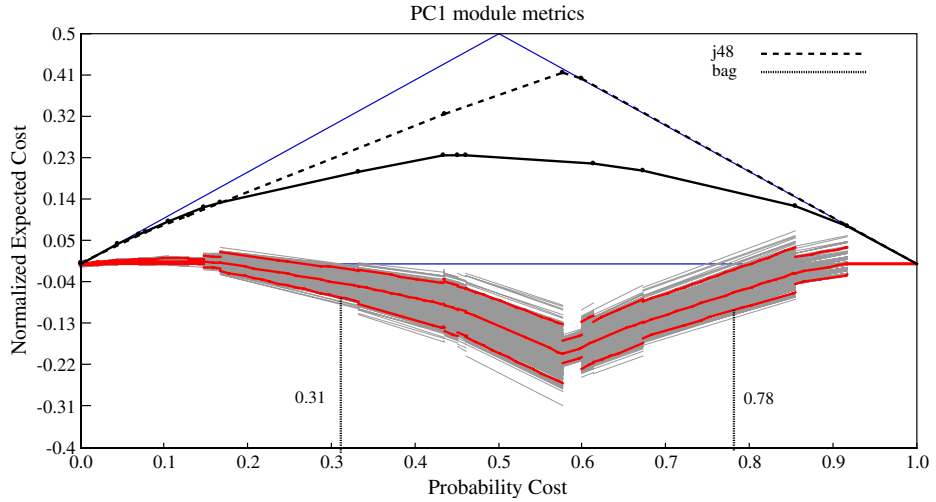


Fig. 14 The 95% confidence interval cost curve evaluation of Bagging and j48 models on PC1

0.78. The confidence bounds span over the entire range of values of probability cost function. Therefore, once we know the misclassification costs which are specific for the project at hand, we can draw conclusions about the statistical significance of differences between models.

Lastly, we compare the models using lift chart analysis. Figure 15 depicts the five *PC1* models, emphasizing the realistic scenario in which up to 40% of software modules will be selected for further analysis. From the lift chart, we can infer the following:

(1) *J48* and *Bagging* overlap when the percentage of software modules to be selected is below *0.05*. *J48* slightly outperforms *bagging* when $x>0.05$.
(2) The performance of all the studied classifiers becomes similar when $x>0.2$.
(3) Around $x=0.15$ (*15%* of modules), *IB1* slightly outperforms *J48*, but *J48* performs better otherwise.

Project PC1 contains *1,109* modules, so a *10%* cut selects *111* modules. If the budget and time restrict verification effort to less than *111* modules, we should choose J48. If the project budget calls for the verification of 166 modules (*15%*), we could choose *IB1* model. However, J48 is the reasonable choice if the verification effort does not surpass 40% of all the PC1 modules.

In summary, given the five models selected for comparison on PC1 project, the outcome is quite complex. Different models address different needs of software projects. If the software quality engineers are confident in the appropriateness of *G-mean* or *F-measure*, the project should choose *IB1* classifier. However, *precision*, *specificity*, *ROC* indices and cost curves all point to the selection of the model generated by *bagging* algorithm. Given the lift chart, the model generated by J48 is the most appropriate, although the *bagging* model performs almost as well.

It is interesting to compare our performance evaluation of PC1 data set with others, for example, the recently published work of Menzies et al. (2007). Authors claim that the best classification performance result, *PD=0.48* and *PF=0.17*, is reached by the specific

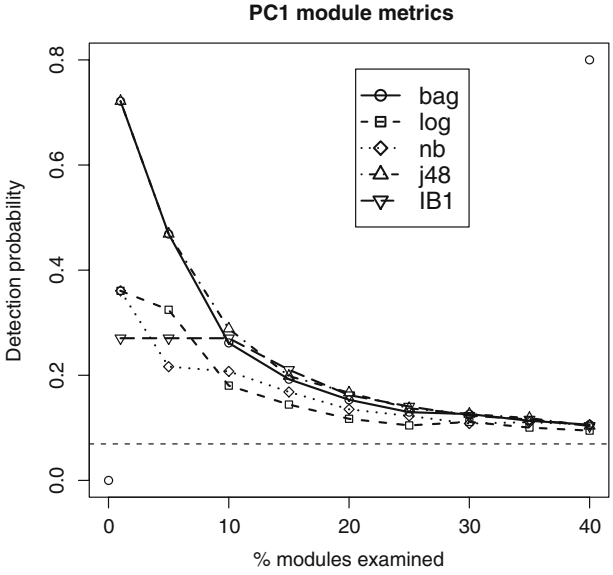


Fig. 15 Lift chart on PC1 module metrics

parameter set-up for Naïve Bayes classifier applied to logarithmically transformed metrics values. These results are comparable to the best (PD, PF) pairs we report in Table 4. But the ROC curves in Fig. 11 reveal that both Bagging and Logistic models offer significantly higher PD rates (above *0.6*) at *PF=0.17*. ROC curve offers to an analyst a range of classification threshold values that are easily overlooked when only selected (PD, PF) pairs are reported. The point we make is that general conclusions about fault prediction models cannot be made by reporting only a few (PD, PF) pairs. The comprehensive evaluation offered here demonstrates that such conclusions should be questioned for validity. This observation is one of the major conclusions of our study.

5.2 KC2 Dataset

The numerical performance indices for project *KC2* are tabulated in Table 6. If we set the minimum *PD* acceptability to 40%, *Naïve Bayes* and *Logistic* classifiers would not be given further consideration, even though the former produced the highest *specificity*, *precision* and *overall accuracy*. The *overall accuracy* of *Bagging* model is the highest, but the other measurements, the *G-mean*, *F-measure, J_coeff* and the distance from the perfect classification are not as good as those generated by J48.

Figure 16 depicts the ROC curves of the five fault prediction models on *KC2*. After observing the *ROC* curves, we infer: 1) The performance of *Bagging*, *Logistic* and *NaiveBayes* models appears similar; 2) These three models are clearly better than *J48* and *IB1*; 3) *J48* appears to be better than *IB1*. The same trend can be observed from the values of *AUC*s in Table 7.

We used the Friedman test and the post-hoc Nemenyi test to analyze whether the difference in the performance of the five models is statistically significant. The average ranks for IB1, J48, Bagging, Logistic, and Naïve Bayes are 4.8, 4.2, 2.6, 1.8, and 1.6, respectively. The outcome of the Nemenyi test at 95% confidence level is shown in Fig. 17. We observe that 1) The performance of Naïve Bayes, Logistic, and Bagging models is statistically indistinguishable; 2) The performance of J48 and Bagging, as well as IB1 and J48 cannot be claimed to be different; 3) All the other comparisons indicate statistically significant differences.

Mean *AUC* values favor the model generated by *Naïve Bayes*. But, maximizing the area under the curve in the left-upper corner of ROC, *AUCa*, favors *Logistic* (see Table 7). Statistical tests tell us that differences between NaiveBayes, Logistic or Bagging models are

Table 6 Performance results for project *MDP-KC2*

Indices	Naïve Bayes	Logistic	IB1	J48	Bagging
PD	0.398	0.389	0.509	**0.546**	0.472
1−PF	**0.950**	0.932	0.858	0.896	0.931
Precision	**0.674**	0.599	0.483	0.578	0.639
Overall Accuracy	**0.836**	0.820	0.786	0.824	**0.836**
$G\text{-mean}_1$	0.518	0.483	0.496	**0.562**	0.549
$G\text{-mean}_2$	0.615	0.602	0.661	**0.700**	0.663
F-measure (β=1)	0.501	0.472	0.496	**0.562**	0.543
F-measure (β=2)	0.434	0.418	0.504	**0.552**	0.498
ED (θ=0.5)	0.427	0.435	0.361	**0.329**	0.377
ED (θ=0.67)	0.492	0.500	0.409	**0.375**	0.433
J_coeff	0.348	0.321	0.367	**0.442**	0.403

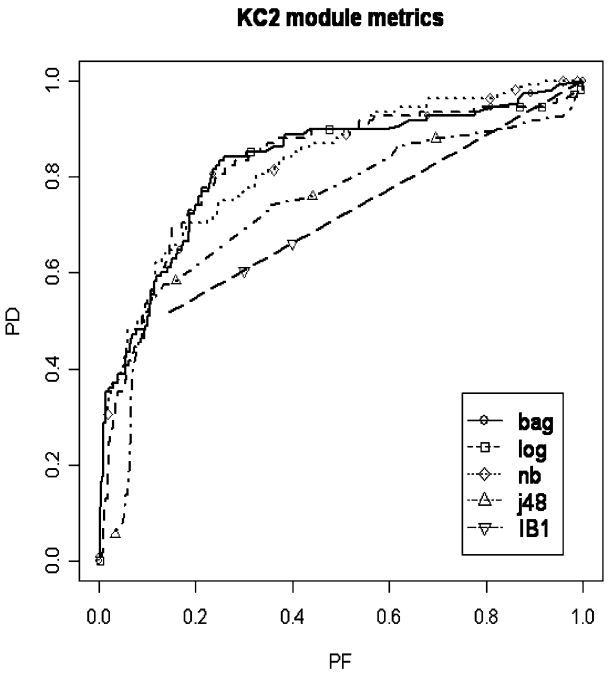


Fig. 16 ROC curves for project KC2

not significant. Given that the tests analyze *AUC* values (rather that *AUCa*) choosing the model developed using *Logistic* classifier is appropriate.

Figure 18 depicts the cost curves of fault prediction models for KC2. The cost curves of *Bagging*, *Logistic* and *Naïve Bayes* are very similar. *IB1* implies the highest cost, followed by *J48*. These observations match those we reached by analyzing the ROC curves and the statistical tests. The vertical line at $PC(+)=0.21$ indicates the performance where the misclassification cost of fault-prone and fault-free classes is the same (*21%* of modules in KC2 are faulty). The region $PC(+)<0.21$ is the verification's cost adverse region and $PC(+)>0.21$ is the risk adverse region. When $PC(+)<0.21$ IB1 is inferior and j48 matches the effectiveness of the trivial classifier. Bagging indicates slight performance advantage in that region. Logistic gains the sight advantage when $0.28<PC(+)<0.89$, but that model becomes worse than the trivial classifier at $PC(+) > 0.88$. Depending on project's actual misclassification cost ratios and verification needs, any of the top three classification models could be selected.

Figure 19 depicts the 95% confidence interval comparison of cost curves from Logistic and Bagging classifiers. The zero line is inside the confidence interval along the entire range of operating points, thus indicating no significant difference between these two classifiers.

Table 7 Mean values of AUC on KC2 models

Dataset	Learner	AUC	AUCa
KC2	IB1	0.689	0.168
	J48	0.729	0.308
	Bagging	0.821	0.472
	Logistic	0.824	**0.484**
	NaiveBayes	**0.832**	0.464

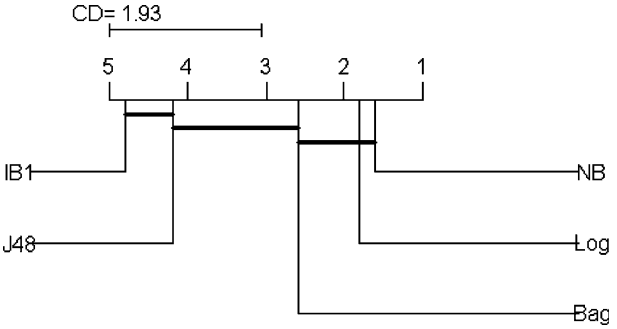


Fig. 17 Comparison of the five classifiers over 10 runs on KC2 data using the Nemenyi test at 95% CI

Further tests between *Bagging, Naïve Bayes, and Logistic*, provide the same result, i.e., no statistically significant differences. B*agging* and *j48*, as one would expect, offer significantly different performance. Figure 20 shows the 95% confidence interval. When PC(+) is in the ranges (0.0, 0.06), (0.31, 0.40), and (0.77, 0.84), performance of bagging and j48 is indistinguishable. Within intervals (0.06, 0.31) and (0.40, 0.77), the confidence interval does not contain the zero line, indicating the significant difference between the two classifiers.

Figure 21 shows lift charts of the five KC2 models. KC2 project contains *523* modules (see Table 1). If we are to select up to *10%* of these modules, Bagging model should be our choice. This may be the most useful subset to be selected if available verification resources are sparse. Logistic is the model of choice if resources allow us to inspect *10*–25% of modules (between *52* and *130*). If KC2 is a safety/mission conscious project and time and budget are not too tight, and more than 25% modules can be analyzed, somewhat surprisingly, J48 emerges as the model of choice.

A brief summary of KC2 model evaluation shows that overall accuracy, precision, and specificity lead towards the selection of Naïve Bayes model. G-means and F-measure, and a specific verification scenario in the lift chart indicate support for J48. From the analysis of ROC and AUC, Naïve Bayes, Logistic, and Bagging are equally good choices. However cost curves or lift charts, identify several special cases when one of the models outperforms the others.

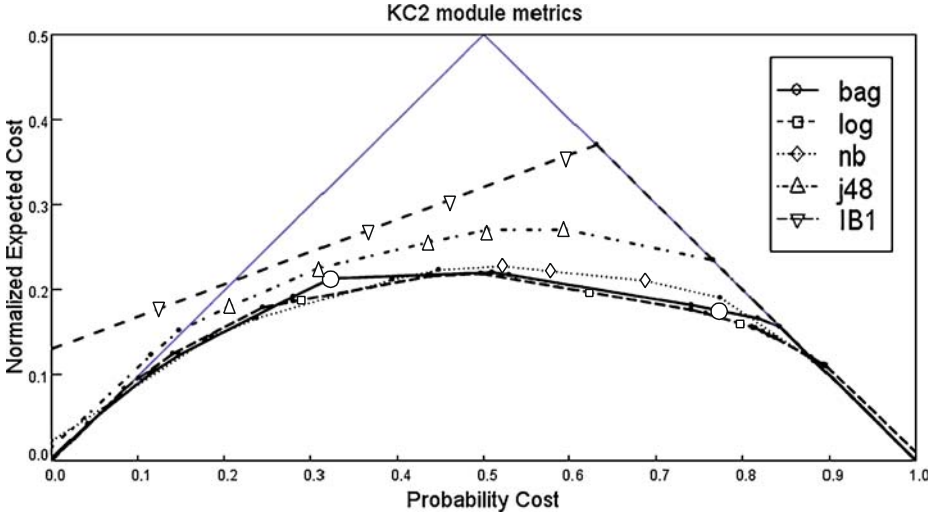


Fig. 18 Cost curves of fault prediction models on KC2

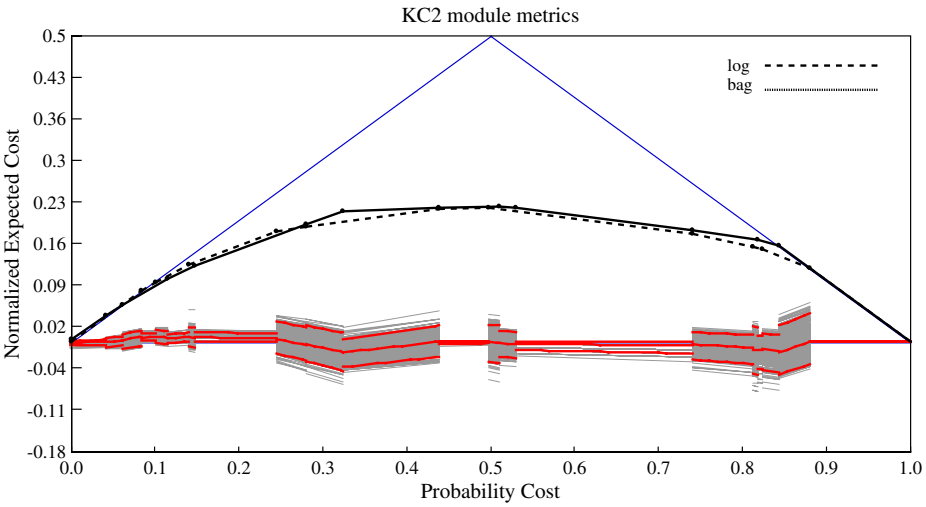


Fig. 19 The 95% confidence interval indicates no significant difference between Bagging and Logistic fault prediction models on KC2 dataset

6 The Guidelines for the Selection of Model Evaluation Techniques

To accurately assess the performance of software quality prediction models, we should carefully choose and interpret the metrics of merit, and evaluate model's performance based on the specific needs of the project. While multiple measurements may be useful for getting a broad understanding of classification performance, they also cause a difficulty when drawing conclusions. As we illustrated, it is not unusual for different performance indices to provide seemingly conflicting comparison results. This phenomenon has been observed

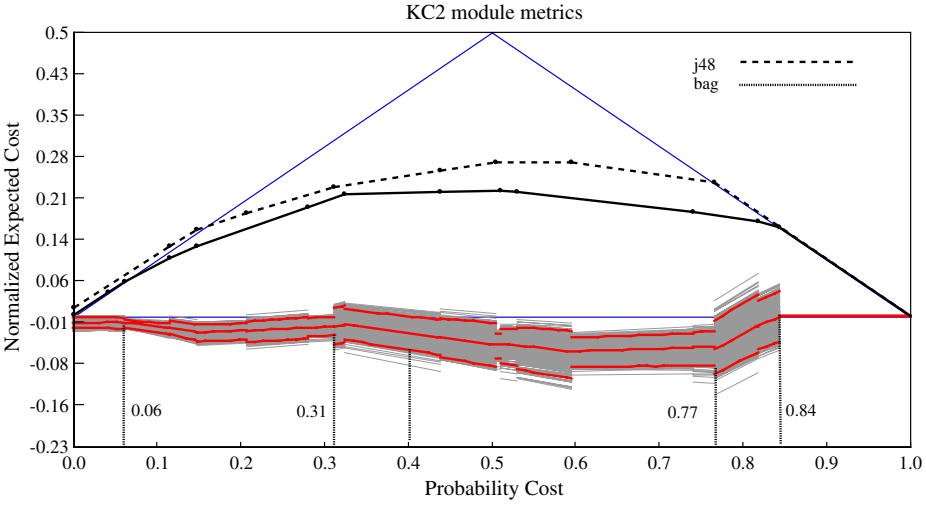


Fig. 20 The 95% confidence interval of the performances of the classifier j48 and bagging on KC2 module metrics

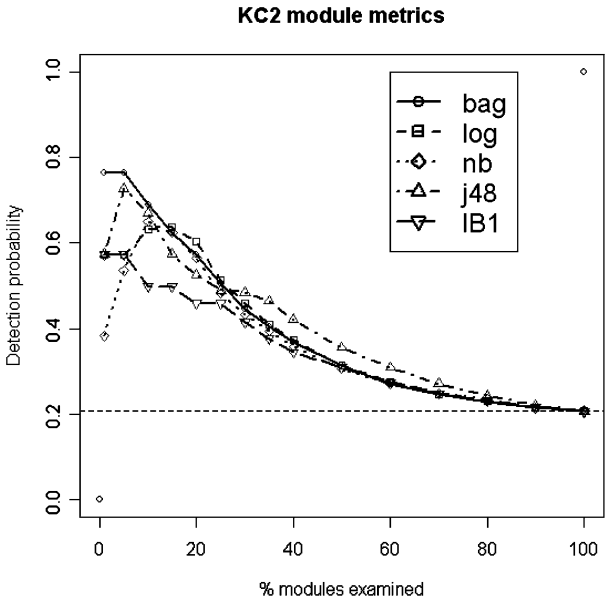


Fig. 21 Lift chart of KC2 module metrics

several times (El-Emam et al. 2001; Kubat et al. 1998; Ohlsson and Alberg 1996). Most performance indices originate in the confusion matrix, making them easy to compute. But in practice, the comparison of models is only meaningful if performance indices can be clearly related to the project specific model evaluation requirements.

In addition to experiments reported in this paper, we conducted many more using more than twenty project datasets available in NASA MDP software engineering repository. We offer the following recommendations that emerged through our work.

- Economic parameters of each project are unique. Classification models in software quality assessment must be developed to address the specific needs of the project. Therefore, analyzing the project's cost characteristics must be the first step in quality assessment. Such analysis builds upon nonfunctional software requirements and results in project-specific modeling requirements and constraints such as misclassification costs, assessment of the resource constraints (how many modules can be exposed to analysis), etc.
- It is unlikely that any single classification algorithm will be able to offer "the best" performance in general. Consequently, a toolbox that supports software quality assessment should include classification algorithms recommended in the empirical software engineering literature. Publicly available machine learning toolsets have enabled a growing consensus in the software engineering publications regarding the selection of the most promising modeling algorithms.
- Overall classification accuracy is not an appropriate measure of classification performance in software fault prediction models due to the fact that fault-free modules typically significantly outnumber the fault prone modules. Sensitivity *(PD)* and the inverse of specificity *(PF)* are much more appropriate for a quick model comparison, but not sufficient for model selection.
- *Sensitivity*, *specificity*, and *precision*, each tell us a unitary story about the performance and should not be used to assess the performance of predictive models

independently. Further, these indices take into account only the specific operational points (selected thresholds) of classification models and may offer misleading conclusions if used to argue model selection.

- The *G-mean*, *F-measure*, *J_coeff* provide more informative insight into model performance because they offer (weighted) combinations of two out of the three indices listed above. However, similar to the comment made above, these coefficients only offer comparison of classification performance for a selected operational point. This is a major limitation as most classification algorithms offer tuning parameters and facilitate a range of operational classification characteristics.
- Graphical model evaluation approaches, such as *ROC* and *PR* curves, offer the advantage of performance evaluation over a range of operational points. However, these are two dimensional plots and model evaluation studies would typically use curve comparisons over different plots.
- Comparison between classification models through *ROC* (or *PR*) curves should focus on the performance points meaningful to the project. For most software projects, the Area Under the Curve of the upper left quadrant of the *ROC (AUCa)* is a more informative measure than the standard *AUC*.
- The limited resources available for verification and validation (*V&V*) are always a concern in software engineering projects. Lift charts facilitate model comparison which supports directing sparse *V&V* resources to software modules which are the most likely to contain faults. Increasing the effectiveness of software *V&V* is the most important goal of quality modeling. Therefore, lift chart analysis typically deserves a prime importance in model comparison.
- Cost curves are the most complex and possibly the most informative charts for model comparison. A cost curve is calculated from many operating points (and classifiers) in the *ROC* space. A cost curve facilitates assessment of prediction models across misclassification cost ratios. Software products operate in environments in which the occurrence of failures implies different consequences. Generally, for most projects it is impossible to state the exact misclassification cost ratio. However, the assessments of failure criticality or fault resolution priority have been a part of software *V&V* activities for a long time. Software development teams typically have a good understanding of the range of pertinent probability cost values. Further, cost curves can also evaluate model performance over the range of operational scenarios in which the proportion of faulty modules changes over time (e.g., after major upgrades) and suggest switching to a more appropriate model. Therefore, we strongly recommend cost curve modeling as one of the most relevant model selection techniques.
- Whenever model selection includes evaluation of several candidates, the application of statistical significance tests is necessary. Such tests support a sound procedure for model selection by distinguishing spurious observations from significant trends in data analysis. Nonparametric statistical tests are usually adequate for comparing software quality models. We recommend the analysis procedure outlined in (Demsar 2006), i.e., the application of the Friedman test to compare the performance of multiple classifiers followed by the Nemenyi post-hoc test.

7 Conclusions

Software quality prediction continues to attract significant attention as it holds the promise for improving the effectives and offering guidance to software verification and validation

activities. Over the past five years, several datasets describing module metrics and their fault content became publicly available. As the result, numerous methodologies for software quality prediction have been proposed in search of "the best" modeling technique. The suggestion to experiment with different modeling techniques followed by the selection of the most appropriate one has become common (Challagulla et al. 2005; Khoshgoftaar et al. 1997; Ohlsson and Alberg 1996). Performance comparison between different classification algorithms for detecting fault-prone software modules has been one of the least studied areas in the empirical software engineering literature. However, fair comparison of models and their evaluation are the prerequisites for model selection. The objective of this paper has been to outline the techniques relevant for specific application requirements of software engineering projects. We believe the approach and evaluation steps outlined in this work have the potential to offer broad understanding, improve relevance, and enhance statistical validity of future studies and experiments.

We surveyed various performance metrics and provided a thorough discussion of adequate and precise evaluation of fault-prediction models in software engineering. An obvious conclusion is that the comparison of fault-prone models is a multi dimensional problem. Rarely will one model or the modeling technique prove to be the best for all possible uses in software quality assessment. Some models will offer advantages when the goal is to select a few modules most likely to be fault-prone. Others will demonstrate superiority in traditional comparison methods for binary classification algorithms, such as *ROC* curves, *PR* curves and areas under these curves, thus offering flexibility and applicability in a range of software engineering project situations. But the reality of software engineering projects cautions us that the overall model classification performance is not the ultimate goal in itself. Rather, optimizing the project cost and maximizing the efficiency of software verification procedures typically tops the agenda. For these purposes, the application of cost sensitive model evaluation indices, such as F-measure and its visual derivatives offers a balanced consideration. In this paper, we described a methodological generalization of cost sensitive numerical performance indices called cost curves. To the best of our knowledge, this is the first attempt to examine the application of cost curves in the software engineering literature. Cost curves offer graphical comparison of model performance across a wide range of module misclassification costs. Being able to characterize the range of misclassification cost ratios is very important because accurate determination of the cost of misclassification is never easy. Therefore, we strongly recommend the use of cost curves in practice and hope they will become a standard tool for software quality model performance evaluation.

Appendix A: Characteristics of NASA MDP dataset

Accurate prediction of fault-proneness in the software development process enables effective identification of modules which are likely to hide faults. Corrective and remedial steps can be adopted early in the development lifecycle before the project encounters costly redevelopment efforts in the later phases. Software projects vary in size and complexity, programming languages, development processes, etc. When reporting a fault prediction modeling experiment, it is important to share the characteristics of the datasets. Here, we outline the characteristics of NASA Metrics Data Prediction (MDP) datasets, which inevitably have a significant impact on fault prediction. Some of our observations seem to be general, i.e., valid across many reported modeling results, while others are specific to this dataset.

First, only a small proportion of software modules are faulty (Menzies et al. 2007) This is one of the most consistent characteristics of software defect databases. Faulty modules constitute only a small portion of the software product base. Table 1 (Section 1.1) provides a sample of projects, released as a part of NASA Software Metrics Data Program (Metrics Data Program NASA IV&V facility, http://mdp.ivv.nasa.gov/). Module faults were detected either during the development or in the deployment. For project *PC1*, only 7% of the modules are faulty and 93% are fault-free; *KC4* has the largest percentage of faulty modules among all five projects considered here, 48%, but it is also the smallest one (only 125 modules) and the only one that uses a scripting language (Perl). In all other datasets, there are at least three times as many fault free modules than faulty ones. Such a significantly skewed distribution of faulty and non-faulty modules in sample datasets presents a problem for supervised learning algorithms typically utilized in software engineering studies. The reason is that most supervised learning techniques aim to maximize the overall classification accuracy and ignore the class distribution. For example, even if we declare all modules in *PC1* as fault-free and misclassify all faulty modules the overall accuracy would achieve 93%. By all means, in the field of predictive models of software fault-proneness, this would be a fantastic result. However, such a model would be useless, as the sole purpose of predictive software quality studies is predicting where faults hide (Menzies et al. 2007).

Second, software metrics used to build predictive models exhibit high correlation (Menzies et al. 2007) Typically, models use software metrics as prediction variables. These metrics tend to correlate with each other. Take the projects listed in Table 1, for example. Each data set contains twenty-one software metrics, which describe the product size, complexity and some structural properties (Metrics Data Program NASA IV&V facility, http://mdp.ivv.nasa.gov/). Tables 8 and 9 display the *Pearson*'s correlation coefficients among the predictive variables for projects *PC1* and *KC2*, respectively. Due to space constraint, we only tabulate the correlation between the lines of code (LOC) and five other randomly chosen variables for each project. In the Table 8, the five randomly selected variables and their metric types are: total operands (TOpnd)—Basic Halstead, volume (V)—Derived Halstead, effort estimate (B)—Derived Halstead, lines of code and comment (LCC)—Line Count, and length (N)—Derived Halstead. The variables selected for *KC2* (Table 9) are: unique operators (UOp)—Basic Halstead, volume (V)—Derived Halstead, design complexity (IV.G)—McCabe, total operators (TOp)—Basic Halstead, and lines of blank (LOB)—Line Count. For both projects, the listed variables are either moderately or strongly correlated; the majority of the correlation coefficients are above 0.90.

The phenomenon of high correlation among predictive variables is termed *multi-colinearity* in regression analysis. As Fenton and Neil (1999) state in (Fenton and Neil 1999), multicolinearity produces unacceptable uncertainty in regression coefficient

Table 8 Correlation coefficients among six predictive variables in project *PC1*

	LOC	TOpnd	V	B	LCC	N
LOC	1.000	0.908	0.937	0.931	0.545	0.924
TOpnd	0.908	1.000	0.976	0.971	0.464	0.996
V	0.937	0.976	1.000	0.995	0.468	0.987
B	0.931	0.971	0.995	1.000	0.468	0.982
LCC	0.545	0.464	0.468	0.468	1.000	0.473
N	0.924	0.996	0.987	0.982	0.473	1.000

Table 9 Correlation coefficients among six predictive variables in project *KC2*

	LOC	UOp	V	IV.G	TOp	LOB
LOC	1.000	0.632	0.986	0.968	0.991	0.909
UOp	0.632	1.000	0.536	0.577	0.615	0.636
V	0.986	0.536	1.000	0.970	0.990	0.887
IV.G	0.968	0.577	0.970	1.000	0.972	0.836
Top	0.991	0.615	0.990	0.972	1.000	0.912
LOB	0.909	0.636	0.887	0.836	0.912	1.000

estimates. Specifically, the coefficients can change drastically depending on which terms (that is, metrics) are present in the model and also depending on the order in which they are placed in the model.

Third, many fault-free modules tend to be small in size The lines of code (LOC) metric is commonly used to measure the size of a module. We calculated the 90th percentile of LOC for faulty and fault-free modules for five MDP projects, and summarized the findings in Table 10. A 90th percentile is a score such that 90% of the scores are below it. For example, for project *PC1*, about 90% of the fault-prone models have at most 114 lines of codes, while 90% of the non-fault modules have at most 47 lines of codes. Except in KC4, fault-free modules tend to be shorter than faulty modules. But many software metrics depend on size and measurements of relatively small modules tend to be "close" to each other. This may confuse machine learning algorithms and is likely to cause poor prediction results. Koru and Liu (2005) made the same observation and stated that "small modules show little variation, which would make it difficult for a machine-learning algorithm to distinguish between small—defective and small—nondefective modules". Dealing with such small components appears to be specific for the MDP dataset studies, limiting the generality of conclusions reached by studying projects included in it.

Fourth, a significant portion of the minority class instances (i.e., fault-prone modules) are "close neighbors" with the majority class instances Table 11 provides the evidence. For each module in a training set, we find the nearest (in Euclidean distance) training set vector (the module with the most similar metrics) and count the proportion of faulty modules whose nearest neighbor belongs to the majority class (i.e., fault-free modules). Besides, we find three nearest neighbors of each faulty module, and compute the percentage of faulty modules that has at least two neighbors (among three) from the majority class. Table 11 shows that a significant number of the minority class cases are close to the majority class instances in the feature space. This phenomenon is the consequence of the previously mentioned characteristics. With the majority of the learning set representing fault-free modules, faulty modules are likely to have their measurements similar to many fault-free ones. With the second and third properties, small modules, faulty or not, are likely similar

Table 10 The 90th percentile of lines of code (LOC) for the collection of faulty modules and fault-free modules in each project

	KC1	KC2	PC1	JM1	CM1	KC4	PC5	MC2
Fault-free	42	55	47	72	55	460	7	68
Faulty	99	167	114	165	131	252	294	134

Table 11 Class membership of the neighborhood instance for the faulty modules

Project	Percent of faulty modules whose nearest neighbor is a majority class instance	Percent of faulty modules that has ≥2 among the three nearest neighbors in the majority class
KC1	66.26	73.62
KC2	58.33	58.33
JM1	67.90	75.46
PC1	75.32	85.71
CM1	73.47	97.96
KC4	31.67	25.00
PC5	59.71	66.94
MC2	57.69	65.38

to each other. This poses a problem for all machine learning techniques (Boetticher 2005), especially for the instance-based learning methods. Project KC4 is an exception and this may be part of the reason why in some studies classification algorithms tend to achieve better overall classification results on this dataset (Menzies et al. 2007) than on others analyzed here.

These four observations are not necessarily the only interesting and/or unique aspects of the MDP datasets. But good understanding of the dataset is necessary for the selection of adequate classification techniques and play important role in model selection and comparison.

Appendix B: A Brief Description of the Six Classification Algorithms

Six machine learning algorithms are used in illustrative examples throughout the paper: Random Forest, Naïve Bayes, Bagging, J48, Logistic Regression and IBk.

Random forest (rf) is a decision tree-based classifier demonstrated to have good performance in software engineering studies by Guo *et al.* (2004). As implied from its name, it builds a "forest" of decision trees. The trees are constructed using the following strategy: The root node of each tree contains a bootstrap sample data of the same size as the original data. Each tree has a different bootstrap sample. At each node, a subset of variables is randomly selected from all the input variables to split the node and the best split is adopted. Each tree is grown to the largest extent possible without pruning. When all trees in the forest are built, new instance(s) is fitted to all the trees and a voting process is taken place. The forest selects the classification with the most votes as the prediction of new instance(s).

Naïve Bayes (nb) "naively" assumes data independence. This assumption may be considered overly simplistic in many application scenarios. However, in software engineering data sets its performance is surprisingly good. Naive Bayes classifier has been used extensively in fault-proneness prediction by (Menzies et al. 2007).

Bagging (bag) stands for bootstrap aggregating. It relies on an ensemble of different models. The training data is resampled from the original data set. According to Witten and Frank (2005), bagging typically performs better than single method models and almost never significantly worse.

J48 is a Weka (Witten and Frank 2005) implementation of Quinlan's C4.5 (Guo et al. 2004) decision tree algorithm. A decision tree is a tree structure where non-terminal nodes represent tests on one or more attributes and terminal nodes reflect decision outcomes. It is a popular and classic machine learning algorithm (Witten and Frank 2005).

Logistic regression (log) is a classification scheme which uses mathematical logistic regression function. The most popular models are generalized linear models.

IBk is the Weka (Witten and Frank 2005) tool implementation of *k*-nearest-neighbor classifier. With *k=1* the default value, *IBk* is in fact *IB1*. This is the basic nearest-neighbor instance based learner that searches for the training instance closest in Euclidean distance to the given test instance and uses the result of the search for classification (Witten and Frank 2005).

References

Adams NM, Hand DJ (1999) Comparing classifiers when the misallocation costs are uncertain. Pattern Recognit 32:1139–1147. doi:10.1016/S0031-3203(98)00154-X

Arisholm E, Briand LC (2006) Predicting fault-prone components in a java legacy system. Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE'06)

Azar D, Precup D, Bouktif S, Kegl B, Sahraoui H (2002) Combining and adapting software quality predictive models by genetic algorithms. 17th IEEE International Conference on Automated Software Engineering. IEEE Computer Society

Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. IEEE Trans Softw Eng 22(10):751–761. doi:10.1109/32.544352

Boetticher GD (2005) Nearest neighbor sampling for better defect prediction. ACM SIGSOFT Software Engineering Notes, 30(4). ACM, New York, NY, pp 1–6

Braga AC, Costa L, Oliveira P (2006) A nonparametric method for the comparison of areas under two ROC curves. International Conference on Robust Statistics (ICORS06). Technical University of Lisbon, 16–21 July 2006, Lisbon, Portugal

Breiman L (2001) Random forests. Mach Learn 45:5–32. doi:10.1023/A:1010933404324

Challagulla VUB, Bastani FB, Yen I-L, Paul RA (2005) Empirical assessment of machine learning based software defect prediction techniques. Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05), pp 263–270

Conover WJ (1999) Practical nonparametric statistics. Wiley, New York

Davis J, Goadrich M (2006) The relationship between precision-recall and ROC curves. Proceedings of the 23rd International Conference on Machine Learning. Pittsburgh, PA, pp 233–240

Demsar J (2006) Statistical comparisons of classifiers over multiple data sets. J Mach Learn Res 7:1–30

Drummond C, Holte RC (2006) Cost curves: an improved method for visualizing classifier performance. Mach Learn 65(1):95–130. doi:10.1007/s10994-006-8199-5

El-Emam K, Benlarbi S, Goel N, Rai SN (2001) Comparing case-based reasoning classifiers for predicting high-risk software components. J Syst Softw 55(3):301–320. doi:10.1016/S0164-1212(00)00079-0

Fenton N, Neil M (1999) Software metrics and risk. The 2nd European Software Measurement Conference (FESMA 99), TI-KVIV, Amsterdam, pp 39–55

Gokhale SS, Lyu MR (1997) Regression tree modeling for the prediction of software quality. In: Pham H (ed) The third ISSAT International Conference on Reliability and Quality in Design. Anaheim, CA, pp 31–36

Guo L, Ma Y, Cukic B, Singh H (2004) Robust prediction of fault-proneness by random forests. Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE 2004), IEEE Press

Khoshgoftaar TM, Lanning DL (1995) A neural network approach for early detection of program modules having high risk in the maintenance phase. J Syst Softw 29(1):85–91. doi:10.1016/0164-1212(94)00130-F

Khoshgoftaar TM, Allen EB, Ross FD, Munikoti R, Goel N, Nandi A (1997) Predicting fault-prone modules with case-based reasoning. The Eighth International Symposium on Software Engineering (ISSRE '07). IEEE Computer Society, pp 27–35

Khoshgoftaar TM, Seliya N (2002) Tree-based software quality estimation models for fault prediction. The 8th IEEE Symposium on Software Metrics (METRICS'02), IEEE Computer Society, pp 203–214

Khoshgoftaar TM, Cukic B, Seliya N (2007) An empirical assessment on program module-order models. Qual Technol Quant Manag 4(2):171–190

Koru AG, Liu H (2005) Building effective defect-prediction models in practice. IEEE Softw 22(6):23–29. doi:10.1109/MS.2005.149

Kubat M, Holte RC, Matwin S (1998) Machine learning for the detection of oil spills in satellite radar images. Mach Learn 30(2–3):195–215. doi:10.1023/A:1007452223027

Lewis D, Gale W (1994) A sequential algorithm for training text classifiers. Annual ACM Conference on Research and Development in Information Retrieval, the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Springer-Verlag, New York, NY, pp 3–12

Ling CX, Li C (1998) Data mining for direct marketing: problems and solutions. Proc. of the 4th Intern. Conf. on Knowledge Discovery and Data Mining, New York, pp 73–79

Ma Y (2007) An empirical investigation of tree ensembles in biometrics and bioinformatics. West Virginia University, PhD thesis, January 2007

Macskassy S, Provost F, Rosset S (2005a) Pointwise ROC confidence bounds: an empirical evaluation. Proceedings of the Workshop on ROC Analysis in Machine Learning (ROCML-2005)

Macskassy S, Provost F, Rosset S (2005b) ROC confidence bands: an empirical evaluation. Proceedings of the 22nd International Conference on Machine Learning (ICML). Bonn, Germany

Menzies T, Stefano JD, Ammar K, Chapman RM, McGill K, Callis P et al (2003) When can we test less? Proceedings of the Ninth International Software Metrics Symposium (METRICS'03), IEEE Computer Society

Menzies T, Greenwald J, Frank A (2007) Data mining static code attributes to learn defect predictors. IEEE Trans Softw Eng 33(1):2–13. doi:10.1109/TSE.2007.256941

Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Trans Softw Eng 31(4):340–355. doi:10.1109/TSE.2005.49

Ohlsson N, Alberg H (1996) Predicting fault-prone software modules in telephone switches. IEEE Trans Softw Eng 22(12):886–894. doi:10.1109/32.553637

Ohlsson N, Eriksson AC, Helander ME (1997) Early risk-management by identification of fault-prone modules. Empir Softw Eng 2(2):166–173. doi:10.1023/A:1009757419320

Selby RW, Porter AA (1988) Learning from examples: generation and evaluation of decision trees for software resource analysis. IEEE Trans Softw Eng 14(12):1743–1757. doi:10.1109/32.9061

Siegel S (1956) Nonparametric statistics. McGraw-Hill, New York

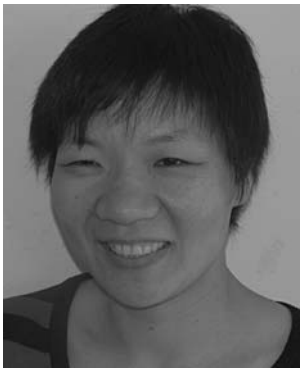
Vuk M, Curk T (2006) ROC curve, lift chart and calibration plot. Metodoloski zvezki 3:89–108

Witten IH, Frank E (2005) Data mining: practical machine learning tools and techniques. Morgan Kaufmann

Youden W (1950) Index for rating diagnostic tests. Cancer 3:32–35. doi:10.1002/1097-0142(1950)3:1<32::AID-CNCR2820030106>3.0.CO;2-3

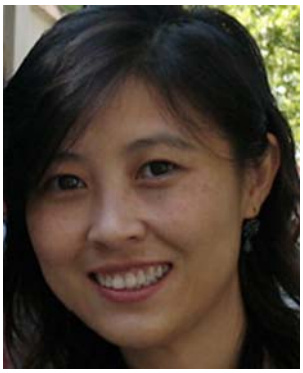
Yousef WA, Wagner RF, Loew MH (2004) Comparison of non-parametric methods for assessing classifier performance in terms of ROC parameters. In Proceedings of Applied Imagery Pattern Recognition Workshop, vol. 33, issue 13–15, pp 190–195

Zhang H, Zhang X (2007) Comments on 'data mining static code attributes to learn defect predictors'. IEEE Trans Softw Eng 33(9):635–637



Yue Jiang is a PhD student in the Lane Department of Computer Science and Electrical Engineering at West Virginia University, where she also earned her MS degree. She received a BS degree in Computer Science in 1993 from Jilin University, Changchun, Jilin province, China. Her research interests include data mining, software quality prediction, and bioinformatics.

Bojan Cukic is a Robert C. Byrd Professor in the Lane Department of Computer Science and Electrical Engineering at West Virginia University, where he also serves as a co-director of the Center for Identification Technology Research, an NSF Industry University Cooperative Research Center. His research interests include software engineering for high-assurance systems, fault-tolerant computing, information assurance, and biometrics. He received a US National Science Foundation Career Award and a Tycho Brahe Award for research excellence from NASA Office of Safety and Mission Assurance. Dr. Cukic is a member of the editorial board of Empirical Software Engineering and a member of the steering committee of IEEE International Symposium on Software Reliability Engineering (ISSRE). He received his MS and PhD degrees in Computer Science from the University of Houston and a BS degree from the University of Ljubljana, Slovenia.



Yan Ma is currently a Senior Research Biostatistician with Bristol-Myers Squibb Company. She was a faculty member in the Department of Statistics at West Virginia University (WVU) from 2004 to 2007, where she taught statistics courses and consulted for the bioinformatics core facility in the Health Sciences Center. Dr. Ma's research interests include bioinformatics, data mining, microarray data analysis, and information integration. She earned her MS degree in statistics and Ph.D. in Computer Science from WVU. She has published in top international conferences and journals in bioinformatics, biometrics and software engineering.