

A Statistical Framework for the Prediction of Fault-Proneness

Yan Ma[†] Lan Guo[‡] Bojan Cukic*

[†]Department of Statistics, West Virginia University, Morgantown, WV 26506
yma@stat.wvu.edu

[‡]MBR Cancer Center/Department of Community Medicine
West Virginia University, Morgantown, WV26506
lguo@hsc.wvu.edu

*Lane Department of Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV 26506
cukic@csee.wvu.edu

Abstract

Accurate prediction of fault prone modules in software development process enables effective discovery and identification of the defects. Such prediction models are especially valuable for the large-scale systems, where verification experts need to focus their attention and resources to problem areas in the system under development. This paper presents a methodology for predicting fault prone modules using a modified random forests algorithm. Random forests improve classification accuracy by growing an ensemble of classification trees and letting them vote on the classification decision. We applied the methodology to five NASA public domain defect data sets. These data sets vary in size, but all typically contain a small number of defect samples in the learning set. For instance, in project PC1, only around 7% of the instances are defects. If overall accuracy maximization is the goal, then learning from such data usually results in a biased classifier, i.e. the majority of samples would be classified into non-defect class. To obtain better prediction of fault-proneness, two strategies are investigated: proper sampling technique in constructing the tree classifiers, and threshold adjustment in determining the winning class. Both are found to be effective in accurate prediction of fault prone modules. In addition, the paper presents a thorough and statistically sound comparison of these methods against ten other classifiers frequently used in the literature.

1 Introduction

Early detection of fault-prone software components enables verification experts to concentrate their time and resources on the problem areas of the software system under development. The ability of software quality models to accurately identify critical components allows for the application of focused verification activities ranging from manual inspection to automated formal analysis methods. Software quality models, thus, help ensure the reliability of the

delivered products. It has become an imperative to develop and apply good software quality models early in the software development life cycle, especially for large-scale development efforts.

The basic hypothesis of software quality prediction is that a module currently under development is fault prone if a module with the similar product or process metrics in an earlier project (or release) developed in the same environment was fault prone [25]. Therefore, the information available early within the current project or from the previous project can be used in making predictions. This methodology is very useful for the large-scale projects or projects with multiple releases.

Many modeling techniques have been developed and applied for software quality prediction. These include, logistic regression [3], discriminant analysis [26, 32], the discriminative power techniques [35], Optimized Set Reduction [10], artificial neural network [27], fuzzy classification [15], Bayesian Belief Networks [16], genetic algorithms [2], classification trees [18, 28, 36, 39], and recently Dempster-Shafer Belief Networks [19]. For all these software quality models, there is a tradeoff between the defect detection rate and the overall prediction accuracy. Thus, a performance comparison of various models, if based on only one criterion (either the defect detection rate or the overall accuracy), may render the comparison only partially relevant. A model can be considered superior over its counterparts if it has both a higher defect detection rate, and a higher overall accuracy. About 65-75% of critical modules and non-fault prone modules were correctly predicted in [23, 26, 28] using the overall accuracy criterion. The decision tree [36] correctly predicted 79.3% of high development effort fault prone modules (detection rate), while the trees generated from the best parameter combinations correctly identified 88.4% of those modules on the average. The discriminative power techniques correctly classified 75 of 81 fault free modules, and 21 of 31 faulty modules [35]. In one case study, among five common classification techniques: Pareto classification, classification trees, factor-based discriminant analysis, fuzzy classification, and neural network, fuzzy classification appears to yield better results with a defect detection rate of 86% [14]. Since most of these studies have been performed using different data sets, reflecting different software development environments and processes, the final judgement on “the best” fault-prone module prediction method is difficult to make.

In this paper, we introduce a novel software quality prediction methodology based on balanced Random Forests [5]. We compare the proposed methodology with many existing approaches using the same data sets. Our approach to predicting fault prone modules runs efficiently on large data sets and it is more robust to outliers and noise compared to other classifiers. Hence, it is especially valuable for the large systems. The prediction accuracy of the proposed methodology is higher as compared to the algorithms available in three statistical and/or data mining software packages, WEKA, See5 and SAS, for the same data sets obtained from NASA. The difference in the performance of the proposed methodology over other methods is statistically significant. As performance comparison between different classification algorithms for detecting fault-prone software modules has been one of the weakest points in software engineering literature, we believe the procedure we outline here, if followed, has a potential to enhance the statistical validity of future experiments.

The remainder of this paper is organized as follows. Section 2 describes Random Forests. Section 3 introduces the balanced Random Forests algorithm. The data sets used in this study are presented in Section 4. Section 5 defines performance measurement parameters used in the experiments. Section 6 presents the experiments, including the methodology, a short description of the classifiers used, and a detailed comparison between the proposed methodology over related work, i.e. logistic regression, discriminant analysis, and classifiers available in two machine learning software packages, See5 [1] and WEKA [40]. Section 7 includes a brief discussion of experimental results. Section 8 summarizes the paper and provides concluding remarks.

2 Random Forests

A classification tree (Breiman *et al.*, 1984) is a top-down tree-structured classifier. It is built through a process known as recursive partitioning whereby the measurement space is successively split into subsets, each of which is equivalent to a terminal node in the tree.

Starting from the root node (i.e., the top node of the tree) which contains the entire sample, all the candidate splits are evaluated independently, the most appropriate one is selected. The “appropriateness” of a split can be evaluated by different measures. The most popular one is based on impurity functions. Impurity measures are the quantification of how well the classes are being separated. In general, the value of an impurity measure is the largest when data are split evenly for attribute values and zero when all the data points belonging to a single class. There are various impurity measures used in the literature. The most commonly used are: *Entropy-based measure* (Quinlan, 1993) and *Gini index* (Breiman *et al.*, 1984). Purity (or homogeneity) of class labels is measured before and after the split. The split which produces the most discrimination between classes is the most appropriate one. The classification tree is grown to a point where the number of instances in the terminal node is small or the class membership of instances in the node is pure enough. The label of the class which dominates the instances in the terminal node is assigned to this node. The fully grown tree may overfit the training data and may need to be cut back by using criteria which balance the classification performance of the tree and the tree’s complexity.

In a tree model, an edge from a parent node to one of its children nodes indicates a rule. This child node is reachable only when the rule is satisfied. A path from the root node to a terminal node of the tree, which consists of at least one edge, specifies a classification rule. Such classification rule is the combination of the functions associated with all of the edges on the path. The decision associated with each classification rule is stated by the label attached to the terminal node. For a new data instance characterized by the input vector \mathbf{x} , we would expose it to the root node of the tree and then follow along the path in which \mathbf{x} satisfies all the rules. Obtained class label at the terminal node represents the classification decision.

Trees represent an efficient paradigm in generating understandable knowledge structures. But, experience shows that the lack of accuracy is the reason that prevents classification trees from being the ideal tool for predictive learning [22]. One way to improve accuracy of classification trees is to utilize ensemble learning. Ensemble methods learn a large number

of models instead of a single model and combine the predictions of all the models with a learning algorithm.

Bagging (Breiman, 1996) and random forests (Breiman, 2001) are ensemble methods. They construct a collection of base models and simply average their predictions. Bagging generally works for different classifiers, including trees or neural networks, while random forests use only trees as base models. Random forests algorithm stems from bagging and can be considered as a special case of bagging [38].

The idea behind bagging is to take a bootstrap replicate (a sample collected with replacement) from the original training set and obtain a model $f(\mathbf{x})$. By bootstrapping, K different versions of the learning set can be generated and K models f_1, \dots, f_K are obtained. Each tree predicts class membership of any test case. For overall classification, the predicted class is determined by plurality voting among the classes C , i.e., the class label most frequently predicted by the K models is selected.

Like bagging, a random forest consists of a collection of K tree classifiers $h_1(\mathbf{x}), h_2(\mathbf{x}), \dots, h_K(\mathbf{x})$. Each classifier is built upon a bootstrap replica of the training set and votes for one of the classes in C . A test instance is classified by the label of the winning class. With bootstrap sampling, approximately 36.8% of the training instances are *not* used in growing each tree (due to sampling with replacement). These data instances are called out-of-bag (OOB) cases. Random forest algorithm uses OOB cases to estimate the classification error and evaluate the performance of the forest.

Besides bootstrap aggregating, another source of randomness called abbreviated bagging is introduced in constructing the forest ensemble. At each node of the tree, the learning algorithm randomly selects and evaluates a subset of m features. The “best” feature subset is chosen as the basis for the decision which splits the node. The value of m is kept fixed at all nodes. In the random forests algorithm implemented in *R* toolset (<http://www.r-project.org>), the default value of m is selected to be the integer nearest to \sqrt{M} , where M is the total number of features in the dataset [6].

Let us denote the joint classifier as $h(\mathbf{x})$. The classification margin function indicates the extent to which the average vote for the correct class y exceeds the average vote for any other class in C . It is defined as:

$$\text{margin}(\mathbf{x}, \mathbf{y}) = P(h(\mathbf{x}) = \mathbf{y}) - \max_{i=1, i \neq \mathbf{y}}^C P(h(\mathbf{x}) = i) \quad (1)$$

The classification margin is estimated from all the tree classifiers in the forest. In the binary classification problem, the margin of an observation is the difference between the proportion of votes for the true class and the proportion of votes for the other classes. The calculated margin lies between -1 and 1 (inclusive). A positive margin indicates a correct classification decision. The larger the margin, the more confidence we may have in the conclusion. If the margin of an observation is negative, then an erroneous decision is made.

The classification error of a forest depends on the strength of individual tree classifiers in the forest and the correlation between them [5, 6, 7].

- *Correlation between tree classifiers in the forest.* No improvement could be obtained

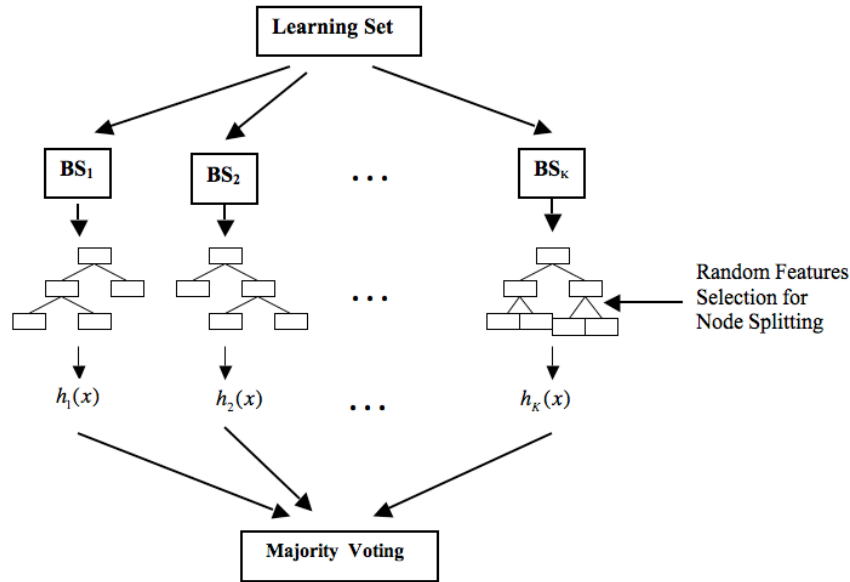


Figure 1: Construction of a Random Forest

by generating identical trees. Two sources of randomness described above, bagging and random feature selection, make the trees look different and therefore lower the correlation between trees. Low correlation lowers the classification error rate.

- *Strength of individual tree in the forest.* Strength of the forest, measured by the average margin over the learning set, relies on the strength of individual tree models. A tree with a low error rate is a strong classifier. Having more diverse strong classifiers in the forest lowers the overall error.

Every tree in a forest of K trees expresses specific classification rules. Given a new instance with feature vector $\mathbf{x} = \{x_1, x_2, \dots, x_P\}$, the classification starts with tree #1 in the forest. The traversal begins from the root and the splitting rule determines the child node to descend to. This process is repeated until the terminal node is reached. The class label attached to the terminal node is assigned to the case. Thus, *tree #1* has made its decision. The classification continues with *tree #2* and follows the same procedure in finding the class label for the same data instance. Upon classifying the instance by every single tree in the forest, the algorithm has K votes, each indicating the likely class to which the data instance belongs to. The class with the majority of votes wins. *Fig. 1* shows the construction of a random forest.

Random forest algorithm overcomes the disadvantages of a single classification tree. Trees are built from different bootstrap samples of the original dataset. Random feature sets used for determining optimal splits at each node stabilize the classifier. Finally, making the

classification decision by voting improves the performance over a single tree. The advantages of random forest outlined in the literature include [5, 6, 34]:

- Easy to use and simplicity.
- Low classification error.
- Robustness with respect to noise.
- Easy handling of large feature spaces.
- Relatively fast tree growing algorithm and the absence of the need for tree pruning.
- Built-in cross validation by using OOB cases to estimate classification error rates.
- High level of predictive accuracy without overfitting.

From a single run of a random forest, we can obtain a wealth of information such as classification error rate, variable importance ranking and proximity measures. Variable importance measures identify attributes which have high ranking in terms of predictive power. Intrinsic proximities provide information on measuring relationship among instances. Detailed description can be found in [6]. Below we describe a few that are important for the application of random forests to the software engineering data sets.

- *Classification Error Rate Estimation.* In random forest, there is no need for cross-validation, i.e., the use of a separate test set to get an unbiased estimate of the classification error [6]. As mentioned earlier, the OOB cases are used for validation. This feature simplifies the design of software engineering experiments, especially in cases when data sets are rather small and the division into a training set and a test set diminishes the statistical representativeness of each of them. OOB testing is performed as follows. OOB cases of the k th tree in the forest are used to test that tree as well as all the other trees in the forest, followed by voting. Comparing the the overall classification with the known class affiliation from the dataset provides an unbiased estimate of the OOB set error rate.
- *Feature Importance Evaluation.* In random forests algorithm, the importance of a variable is defined in terms of its contribution of predictive accuracy. The implementation of random forest algorithm in R offers users two importance measures. The first importance measure, “Mean Decrease in Accuracy” is defined through OOB testing. Randomly permuting the values of the i th variable in the OOB set of each tree results in a permuted OOB set. Random forest is run on such a permuted OOB set. The decrease in the margin resulting from random permutation of the values of i th variable averaged across all the cases in the OOB set produces the importance measure for this variable. The second measure, “Mean Decrease in Gini”, is based on the *Gini* index. If a split of a node is made on the i th variable, as cases in the children nodes become more homogeneous, the *Gini* impurity criterion for the children nodes should be lower

than that measured in the parent node. The sum of all decreases in impurity in the forest due to a variable, normalized by the number of trees, forms a variable importance measure. When software engineering metrics are used to predict quality attributes, ranking the importance of measurements is very important because it helps explain the dependencies in the development life-cycle. Having importance measures incorporated in the classification toolset further simplifies the experimental design.

- *Proximity Measurements.* The proximity measurements provide intrinsic measures of similarities between cases. After each tree is built, the entire training set is evaluated by this tree. If two cases, a and b , fall in the same terminal node, then their proximity measure $prox(a, b)$ is increased by one. At the end of the forest construction, proximity measures are normalized by the number of trees. These measures can be used in missing value replacement, where a missing value is estimated by a proximity weighted sum over the available attribute values. As software engineering measurements are imperfect, missing attribute values are quite common, making this feature of random forests appealing. Proximity measurements can also serve as similarity scores and then be used in clustering.

3 Balanced Random Forests

In many applications including the prediction of fault prone modules in software engineering, the prediction models are faced with class imbalance problem. It occurs when the classes in C have a dramatically different numbers of representatives in the training dataset and/or very different statistical distributions. Learning from imbalanced data can cause the classifier to be biased. Such bias is the result of one class being heavily over-represented in the training data compared to the other classes. Classes containing relatively few cases can be largely ignored by the learning algorithms because the cost of performing well on the large class outweighs the cost of doing poorly on the much smaller classes, provided that the algorithm aims at maximizing overall accuracy [11]. For instance, a binary classification problem such as the identification of fault prone modules may be represented by 1,000 cases, 950 of which are negative cases (majority class, not fault prone modules) and 50 are positive cases (minority class, fault prone modules). Even if the model classified all the cases as negative and misclassified all the positive cases, the overall accuracy could reach 95%, a great result for any machine learning classification algorithm. In many situations, the minority class is the subject of our major interest. In practice, we want to achieve a lower minority classification error even at the cost of a higher majority class error rate. For example, if the goal of identifying fault prone modules early in software development is exposing them to a more rigorous set of verification and validation activities, the imperative is to identify as many potentially faulty modules as possible. If we happen to misclassify non-faulty modules as faulty, the verification process will increase its cost as some modules are unnecessarily analyzed. But, the consequence of misclassifying faulty software as non-faulty may be a system failure, a highly undesirable outcome.

In the random forest algorithm described in Section 2, every single tree is built upon

a bootstrapped sample from the original learning set. In cases when classes represented in the original learning set are unbalanced, the bootstrapped sample is also unbalanced. Consequently, learning from such data would result in a biased classifier. To make use of all available information in the training data and avoid producing a biased classifier, a proper sampling technique needs to be introduced into the random forests algorithm.

One way is to randomly partition the majority class S_{maj} into subsets $S_{maj\ 1}$, $S_{maj\ 2}$, \dots , $S_{maj\ k}$ of (approximately) equal size. The number of subsets k depends on the size of the minority class, S_{min} , since the number of cases in $S_{maj\ i}$ should be comparable with the number of cases in S_{min} . For instance, software project CM1 (see *Table 1*) consists of 449 non-defective modules and 49 faulty modules. We could randomly partition the large class into 9 subsets with each subset containing roughly 50 cases. Combining each $S_{maj\ i}$ with S_{min} results in k balanced samples. Let’s call them “communities”. We could then run random forests algorithm on each community. The final classification decision would be made by plurality voting among k communities.

The other modification of the conventional random forest algorithm could be to build each tree in the forest on a balanced sample. Instead of taking a bootstrapped sample from the entire learning dataset, each single tree is build upon a set which mixes the two class samples of approximately the same size: one is a bootstrapped sample from S_{min} and the other sample is taken randomly with replacement from S_{maj} [12]. The size of the balanced set from which a tree is grown is dependent on the size of minority class. Take the project CM1, for example. Each tree would be built on a balanced sample with 49 fault-prone modules and a subset of non-defective modules of approximately the same size. As the number of the trees in the forest becomes large, all the instances in the original learning set tend to be selected to build the forest. Trees can be grown to maximum size without pruning. As in the conventional random forest, only a random subset of features are selected to make the splitting. We implemented this “balanced tree” feature in the random forests package available from R toolset. This feature has been used in the conduct our experiments, in addition to traditional random forests algorithm.

4 Experimental Database

The defect data sets used in our case studies have been collected from mission critical projects at NASA, as a part of the Software Metrics Data Program (MDP). *Table 1* provides the simple description of the characteristics of software projects which served as the origin of the metrics data sets. Basic data sets properties such as sample size, proportion of defects, instances and data pre-processing methods are described in *Table 2*. Only the data set describing project JM1 contains some missing values. These were removed prior to the statistical analysis. Each data set contains twenty-one software product metrics which describe product’s size, complexity and some structural characteristics [4] (see *Table 3*). Each module in this collection is measured in terms on the same software product metrics. A class label is associated with each module, indicating if the module is defect-free or defects were detected during the development or in the deployment. From *Table 2*, we can see that the fault-prone modules

Table 1: Projects Description

Project	Source Code	Description
KC1	C++	Storage management for receiving/processing ground data
KC2	C++	Science data processing
JM1	C	Real-time predictive ground system
PC1	C	Flight software for earth orbiting satellite
CM1	C	A NASA spacecraft instrument

Table 2: Data Sets Characteristics

Project	# of Instances	Pre-processing	% Defects
KC1	2,109	-	15%
KC2	523	-	21%
JM1	10,885	missing values removed	19%
PC1	1,109	-	7%
CM1	498	-	10%

constitute only a small portion of the data sets. Therefore, the representation of the classes in the learning set does not reflect their importance in the problem.

5 Performance Measurement Indices

Figure 2 represents information about the possible outcomes, actual and predicted classification, for any two class classifier. Several standard terms have been defined for this two class matrix. In our study, we call the software modules with defects “positive” cases, while the modules without defects are termed “negative” cases. In order to be able to evaluate how well classification algorithm performs, we define several performance metrics below.

The *PD*, the *Probability of Detection or recall* is the percentage of fault-prone modules that are correctly predicted by the classification algorithm. It is defined as:

$$PD = \frac{TP}{TP + FN} \quad (2)$$

The *PF* (*Probability of False Alarm*) is the proportion of defect-free modules that were erroneously classified as fault-prone, calculated as:

$$PF = \frac{FP}{FP + TN} \quad (3)$$

The *ACC* (*Accuracy*) is the total number of modules that were predicted correctly. It is calculated as follows:

Table 3: Metric Descriptions of Five Data Sets

Metric Type	Metric	Definition
McCabe	$v(G)$	Cyclomatic Complexity
	$ev(G)$	Essential Complexity
	$iv(G)$	Design Complexity
	LOC	Lines of Code
Derived	N	Length
Halstead	V	Volume
	L	Level
	D	Difficulty
	I	Intelligent Count
	E	Effort
	B	Effort Estimate
	T	Programming Time
Line Count	LOCcode	Lines of Code
	LOComment	Lines of Comment
	LOBlank	Lines of Blank
	LOCcodeAndComment	Lines of Code and Comment
Basic Halstead	UniqOp	Unique Operators
	UniqOpnd	Unique Operands
	TotalOp	Total Operators
	TotalOpnd	Total Operands
Branch	BranchCount	Total Branch Count

		Defect Predicted?	
		No	Yes
Module with Defect?	No	True Negative (TN)	False Positive (FP)
	Yes	False Negative (FN)	True Positive (TP)

Figure 2: Confusion Matrix of Defect Prediction

$$ACC = \frac{TP + TN}{TN + FP + FN + TP} \quad (4)$$

The *True Negative Rate (TNR)* is the proportion of correctly identified defect-free modules. It is calculated as:

$$TNR = \frac{TN}{TN + FP} = 1 - PF \quad (5)$$

The *precision* is proportion of correctly predicted fault-prone modules, calculated as follows:

$$Precision = \frac{TP}{TP + FP} \quad (6)$$

In our case studies, the number of fault-prone modules is much smaller than the number of non-defective modules. Therefore, the accuracy is a good measure of performance [29]. *Figure 3* shows the margin plot (please review Section 2 for the definition of “margin”) for projects PC1 and KC2 using the random forest algorithm (majority voting). Since maximizing overall accuracy is the goal when the traditional random forest algorithm is applied, a good overall accuracy measures were obtained. However, a large proportion of fault-prone modules (cases belonging to minority class) were misclassified. The performance measures that take the “imbalance of data” into account when assessing classification success include *geometric mean (G-mean)* [29] and *F-measure* [30], defined as follows:

$$G - mean_1 = \sqrt{PD * Precision} \quad (7)$$

$$G - mean_2 = \sqrt{PD * TNR} \quad (8)$$

$$F - measure = \frac{(\beta^2 + 1) * Precision * PD}{\beta^2 * Precision + PD} \quad (9)$$

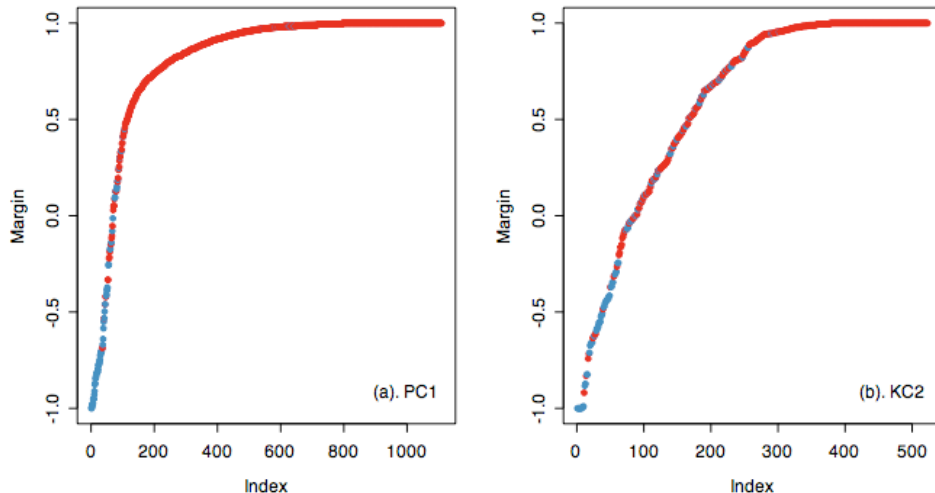


Figure 3: Margin plot: a graphical representation of the confusion matrix. The horizontal axis are case index. Cases from majority class and minority class are represented by red dots and blue dots, respectively. A positive margin associated with a case indicates a correct decision while a negative margin means an erroneous decision.

In (9), β takes any non-negative value and is used to control the weight assigned to PD and $precision$. If we set β to 1, then equal weights are given to PD and $precision$. If all the positive cases are predicted incorrectly, then both G -mean's and F -measure result in a measure of 0. In this paper, we assign equal weights to PD and $precision$ to calculate F -measure.

Receiver Operating Characteristic (ROC) curves can also be used to evaluate the performance of classifiers. An ROC curve plots PF against PD and provides a visual tool for examining the tradeoff between the ability of a classifier to correctly identify positive cases and the number of negative cases that are incorrectly classified. The higher the PD at low PF values (i.e., high y-axis values at low x-axis values) the better the model. A numerical measure of the accuracy of the model can be obtained from the area under the curve, where an area of 1.0 suggests near perfect accuracy, while an area of less than 0.5 shows that the model is worse than simple random guessing [20].

6 Experiments

In this study, we first compare the classification performance of the traditional random forest algorithm with a set of diverse but well known classifiers. We report various performance measures. Then, the performance of the balanced random forest classifier is investigated with respect to the conventional random forests algorithm.

6.1 Methodology

Having grown a forest of decision trees using the random forests algorithm, given a new instance of a software module, each tree in the forest casts a vote and makes its own classification decision. Consequently, a proportion of the trees vote the module as “fault-prone” and the rest classify it as “not-fault-prone”. In binary classification problems using majority voting rule, if at least 50% of the trees reach an agreement such agreement is considered the final decision. Random forests, aiming at the minimization of the overall error rate, keep the error rate low in the majority class and possibly high in the minority class. This obviously causes a problem for software quality prediction since many possibly flawed modules will be misclassified as fault-free and released into the later phase of the software life cycle without proper verification and validation checks.

Imbalance makes classification error lean towards the minority class. Random forests can adjust for imbalance by altering the voting thresholds. User-specified voting cutoffs make the random forest algorithm flexible. If user-defined voting thresholds are used, the “winning” class is the one with the maximum ratio of proportion of votes to cutoff. The default cutoff is $1/c$ where c is the number of classes (i.e., majority vote wins). The fault proneness problem in software engineering has binary predictive outcomes (“fault-prone” and “not-fault-prone”), so the cutoff is a vector of length two $\{c_{fp}, c_{nfp}\}$, with the condition that $c_{fp} + c_{nfp} = 100\%$. With a module to be classified, suppose p_{fp} is the proportion of the trees in the forest that vote for “fault-prone” and $p_{nfp} = 1 - p_{fp}$ in favor of “not-fault-prone”, then the module is predicted to be fault-prone if $\frac{p_{fp}}{c_{fp}} > \frac{p_{nfp}}{c_{nfp}}$. Under the constraints that both c_{fp}, c_{nfp} and p_{fp}, p_{nfp} sum up to 1, as long as $p_{fp} > c_{fp}$, the module is identified as fault-prone.

In the experiment with the traditional random forests algorithm, we use a sequence of user defined thresholds between 0 to 1 (exclusive), with the increment of 0.02, as c_{fp} . Consequently, c_{nfp} takes the value of $1 - c_{fp}$. Software quality engineers who conduct these type of studies can decide to choose a threshold value that produces the best classification results for their specific projects. By changing the threshold of voting, different combination of PD and PF can be obtained. The ROC curves can thus be constructed. This is one way of addressing the imbalance problem in software engineering data sets

The alternative solution of the problem caused by data imbalance is to use the balanced random forests approach, discussed in Section 3. To remind readers, this approach call for the base classifiers (trees) in the ensemble to be built from a balanced sample. A simple majority voting thresholds are then used by the balanced random forests classifier and the corresponding ROC curve represents its performance characteristics. In our experiments with balanced random forests, the area under the ROC curve is calculated using the trapezoidal rule [21].

It has been shown using the large number theory that random forests are not overfit the dataset from which they learn the theory [5]. The classification error rate fluctuates at first and then converges as additional trees are added to the forests. In our experiments, we constructed forests of 800 trees.

6.2 Comparing Classification Results

The performance of the traditional random forest algorithm using variable user defined thresholds is first compared with statistical methods such as logistic regression, discriminant function analysis, a single classification tree, boosting, etc. Some of these methods are implemented in See5 (See5, 2005), some of them in WEKA, an open source software issued under the GNU public license (Witten and Frank, 1999) and some in SAS (SAS, 2005). We used 10-fold cross-validation to evaluate the prediction accuracy of the algorithms whose classification performance we compared with random forests. The 10-fold cross-validation was run for at least 10 times in each experiment on the original data sets. The result with the least variance was chosen as the final result. Below we briefly describe the classifiers included in our study.

- *Logistic Regression.* Binary logistic regression is a form of regression which is used when the dependent is a dichotomy and the variables are of any type. Logistic regression applies maximum likelihood estimation after transforming the dependent into a logit variable. In this way, logistic regression estimates the probability of defect occurrence in a software module. The LOGISTIC procedure in SAS was used in our experiments.
- *Discriminant Analysis.* Discriminant analysis is a very useful statistical tool. It takes into account the different variables of an instance and works out which class the instance most likely belongs to. Linear discrimination is the classification algorithm most widely used in practice. It optimally separates two groups, using the Mahalanobis metric or generalized distance. It also gives the same linear separation decision surface as Bayesian maximum likelihood discrimination in the case of equal class covariance matrices. In our experiments, the DISCRIM procedure from SAS was used on the datasets describing five NASA projects.
- *Classification Tree.* This is one of the classifiers included in the commercial tool See5 . It builds a tree-structured model for classification.
- *Boosting.* Like random forests, boosting is also an ensemble methodology. Base models are constructed sequentially on modified versions of the data with the modification being that the weights of the cases are re-evaluated in order to emphasize misclassified cases. The predictions of the models are integrated through a weighted voting [13]. We used the boosting algorithm's implementation from WEKA.
- *RuleSet.* Rulesets are classifiers generated by See5. Rulesets contain an unordered collections of if-then rules. Each rule consists of one or more conditions that must all be satisfied if the rule is to be applicable. There is also a class label associated with each rule.
- *J48.* This non-parametric machine learning algorithm is implemented in WEKA. J48 is based on Quinlan's C4.5 algorithm [33] for generating decision trees.

- *IBk*. IBk is a WEKA classifier which implements the k-nearest neighbor (*KNN*) algorithm. To classify a new instance, the *KNN* rule chooses the class that is the most common among this new case’s *k* neighbors “closest in distance” in the learning set. IB1 is *KNN* with $k = 1$.
- *VF1*. VF1 is a WEKA classifier implementing the voting feature interval classifier. For numeric attributes, upper and lower boundaries (intervals) are constructed around each class. Discrete attributes have point intervals. Class counts are recorded for each interval on each attribute. Classification is made by voting.
- *Naive Bayes*. Naive Bayes learner is a classifier implemented in WEKA. It is based on probability models that incorporate strong independence assumptions. Naive Bayes classifiers can handle an arbitrary number of independent variables, whether continuous or categorical. Given a set of variables, $\mathbf{X} = \{X_1, X_2, \dots, X_p\}$, Naive Bayes constructs the posterior probability for the class c_j among a set of possible classes in C ,

$$P(c_j|\mathbf{X}) = P(c_j) \prod_{i=1}^p P(X_i|c_j) \quad (10)$$

Naive Bayes classifier calculates the above conditional probability for each class in C and the predicted class is the one with the highest probability.

- *Kernel Density*. This simple kernel density classifier is implemented in WEKA. A detailed description of kernel-based learning can be found in [37].
- *Voted Perceptron*. This WEKA classifier implements the voted perceptron algorithm by [17]. A list of all prediction vectors is generated after each and every classification error during training. For each such vector, the algorithm counts the number of iterations it “survives” until the next classification error is made. This count is treated as the “weight” of the prediction vector. To make a prediction Voted Perceptron algorithm computes the binary prediction of each prediction vector and combines all the predictions using a weighted majority vote. The weights used are the survival times described above. This makes sense since good predictions vectors tend to survive for a long period of time and, thus, have higher weight in the majority vote.
- *Hyper Pipes*. HyperPipe is a WEKA classifier. For each class, a HyperPipe is constructed that contains all points of that class. The algorithm, essentially, records the attribute bounds observed for each class. Test instances are classified according to the class that mostly contains the attributes of the new instance.
- *Decision Stump*. DecisionStump builds simple binary decision “stumps” for classification problems. “Stumps” are tree models which limit growth to just two terminal nodes. This method copes well with missing values by treating “missing” as a separate attribute value. DecisionStump is mainly used in conjunction with the logitboost method and it is implemented in WEKA.

- *KStar*. KStar [24] is an instance-based classifier. The class of a test instance is based upon the class of those instances in the training set similar to it, as determined by some similarity measurement. The underlying assumption of this type of classifiers is that similar cases will belong to similar classes.
- *ROCKY*. ROCKY is a defect detector toolset used in experimental selection of modules for software inspection at NASA IV&V facility in Fairmont, West Virginia (Menziez et al., 2003). ROCKY detectors are built by exhaustively exploring all singleton rules of the form: $attribute \geq threshold$, where *attribute* is every numeric attribute present in the data set, and *threshold* is a certain percentile value of the corresponding attribute. ROCKY has been applied to predicting fault prone modules in projects KC2 and JM1. Predictions based on individuals metrics were presented in [31].

The above classifiers have been chosen for inclusion in our case study either because they have been applied in past in one of the related software engineering experiments or because machine learning literature recommends them as robust tools for two-class classification problems.

6.3 Experimental Results: Traditional Random Forests Algorithm

In this section, we report the comparison of classification performance among the above mentioned tools and techniques on the five NASA software engineering data sets. We discuss the performance measures such as *PD*, *ACC*, *Precision*, $G - mean_1$, $G - mean_2$ and $F - measure$. The comparison results are summarized in *Tables 4 - 8*. Since both $G - mean$ and $F - measure$ are more appropriate for assessing the performance on the imbalanced data sets, in the tables we highlight the highest three values of $G - mean$ and $F - measure$ across all the classifiers under study. Due to space constraints, not all the results from traditional random forest algorithm are shown here. Only the classification outcomes associated with the three cutoffs: (0.9, 0.1), (0.8, 0.2) and (0.7, 0.3) are listed in the tables. By adjusting the thresholds in random forests, the algorithm tends to focus more on the accuracy of the minority class while trading off accuracy in the majority class. The performance of balanced random forests algorithm is reported in the next subsection.

As the classification results vary across different NASA data sets, we present them for each data set first and then generalize the observations later in the discussion.

JM1 Project. JM1 is the largest data set in our study. It contains software metrics and fault-proneness information on 10,885 modules. *Table 4* shows that random forests outperform almost all the other classification techniques. Random forests at cutoffs (0.8, 0.2) and (0.7, 0.3) achieve the highest values in both $G - mean$ and $F - measure$. Logistic regression produces an acceptable value in $G - mean_2$, but a low result in $F - measure$, while discriminant function analysis has the third highest $F - measure$, but a relatively low $G - mean$ scores. Many classifiers used on this project achieve accuracy of up to 81% but are associated with very low probability of detection. HyperPipes, on the other hand, beats all the other methods in probability of detection, but the overall accuracy is not adequate.

JM1 data set is known to suffer from noisy software engineering measurements and it is not surprising that random forests outperform other classifiers on a large and noisy data set.

PC1 Project. PC1 is a fairly large data set with 1,109 modules. Random forests algorithm shows clear advantage over the other methods (see *Table 5*). Although classifier VF1 obtains a superior result on *PD*, the other performance measures prevent it from being a candidate model for software quality prediction. The instance-based classifier IB1 achieves an acceptable *F-measure* compared with other methods except for random forest, but its *G-mean* values are low. Some of the prediction models such as RuleSet, Boosting and VF1 either have a high value in *PD* or accuracy, but not both, therefore can not be used in software quality prediction.

KC1 Project. KC1 project reports the measurements on 2,109 software modules. Generally speaking, random forests surpass almost all the compared methods (see *Table 6*) in terms of *G-mean* and *F-measure*. *KStar* produces comparable results in *G-mean₁* and *F-measure*, but worse in *G-mean₂*. Logistic regression has a relatively high *G-mean₂*, but achieves low value in *F-measure*. Classifiers such as single classification tree, RuleSet and boosting achieve accuracy up to around 85%, but the *PD* is not acceptable. VF1, on the other hand, has an impressive *PD*, but the accuracy and precision measures are not satisfactory.

KC2 Project. KC2 contains software measurements of 520 modules. *Table 7* compares the performance of random forests with ten other classification models on the project KC2. Clearly, random forest at cutoff (0.8, 0.2) produces the best result based on *G-mean* and *F-measure*. Discriminant function analysis also provides decent performance measures. It seems that except for the VotedPerceptron, most other classifier achieve comparable results.

CM1 Project. CM1 is the smallest data set in our study. Only 498 modules are included. From the results in *Table 8*, there is no significant advantage of random forests over the compared methods. Logistic regression and discriminant function analysis appear to perform well on project CM1 based on *G-mean* and *F-measure*. NASA quality engineers noted that this is probably the most straightforward data set for predicting fault-prone modules. It appears that the strengths of random forests algorithm are not reflected in better performance on CM1.

6.4 Comparison between Traditional and Balanced Random Forests

The next step in our study is to compare the performance of balanced random forests (BRF) with the traditional random forests (RF) algorithm. An appropriate way to compare the performance of these two variants of the same classification algorithm is by using ROC curves to describe the results of their application to the same data sets. The results of our first experiment suggest that the traditional RF algorithm is the most suitable choice for the classification tool for most NASA data sets. Our goal here is to investigate whether the application of BRF can bring additional tangible benefits to the prediction of fault-prone modules in these data sets.

The ROC curves in Figure 4 indicate that the performance differences between RF and

Table 4: Performance Comparison on Project JM1

Methods	PD(Recall)	ACC	Precision	$G - mean_1$	$G - mean_2$	F-measure
Logistic	0.654	0.658	0.315	0.454	0.656	0.425
Discriminant	0.509	0.736	0.368	0.433	0.634	0.427
Tree	0.131	0.811	0.546	0.268	0.357	0.211
RuleSet	0.115	0.810	0.540	0.249	0.335	0.190
Boosting	0.118	0.808	0.515	0.246	0.339	0.192
KernelDensity	0.194	0.810	0.523	0.319	0.431	0.283
NaiveBayes	0.197	0.803	0.477	0.306	0.432	0.279
J48	0.247	0.802	0.476	0.343	0.481	0.325
IBk	0.369	0.761	0.379	0.374	0.562	0.374
IB1	0.376	0.756	0.371	0.373	0.564	0.373
VotedPerceptron	0.604	0.560	0.243	0.383	0.576	0.347
VF1	0.868	0.418	0.232	0.448	0.519	0.366
HyperPipes	1.000	0.195	0.194	0.440	0.046	0.324
ROCKY	0.338	0.752	0.352	0.345	0.536	0.345
RF(cutoff=(0.9,0.1))	0.827	0.557	0.281	0.482	0.638	0.419
RF(cutoff=(0.8,0.2))	0.645	0.696	0.346	0.472	0.676	0.450
RF(cutoff=(0.7,0.3))	0.483	0.784	0.445	0.463	0.643	0.463

Table 5: Performance Comparison on Project PC1

Methods	PD(Recall)	ACC	Precision	$G - mean_1$	$G - mean_2$	F-measure
Logistic	0.234	0.885	0.208	0.221	0.467	0.220
Discriminant	0.429	0.849	0.211	0.301	0.615	0.283
Tree	0.221	0.929	0.476	0.324	0.466	0.302
RuleSet	0.177	0.932	0.531	0.307	0.418	0.265
Boosting	0.130	0.940	1.000	0.361	0.361	0.230
J48	0.247	0.934	0.556	0.370	0.493	0.342
IB1	0.416	0.914	0.389	0.402	0.629	0.402
KStar	0.273	0.921	0.399	0.330	0.514	0.324
NaiveBayes	0.299	0.887	0.244	0.270	0.528	0.269
VF1	0.883	0.193	0.071	0.251	0.353	0.132
RF(cutoff=(0.9,0.1))	0.740	0.823	0.245	0.426	0.784	0.368
RF(cutoff=(0.8,0.2))	0.506	0.908	0.379	0.438	0.689	0.433
RF(cutoff=(0.7,0.3))	0.442	0.934	0.531	0.484	0.655	0.482

Table 6: Performance Comparison on Project KC1

Methods	PD(Recall)	ACC	Precision	$G - mean_1$	$G - mean_2$	F-measure
Logistic	0.752	0.711	0.317	0.488	0.727	0.446
Discriminant	0.638	0.790	0.390	0.499	0.722	0.484
Tree	0.193	0.848	0.523	0.318	0.432	0.282
RuleSet	0.187	0.852	0.564	0.325	0.427	0.281
Boosting	0.169	0.862	0.732	0.352	0.409	0.275
KStar	0.509	0.855	0.532	0.521	0.684	0.520
VF1	0.957	0.195	0.156	0.387	0.231	0.269
RF(cutoff=(0.9,0.1))	0.844	0.711	0.330	0.528	0.761	0.475
RF(cutoff=(0.8,0.2))	0.700	0.782	0.387	0.520	0.747	0.498
RF(cutoff=(0.7,0.3))	0.561	0.825	0.447	0.501	0.700	0.498

Table 7: Performance Comparison on Project KC2

Methods	PD(Recall)	ACC	Precision	$G - mean_1$	$G - mean_2$	F-measure
Logistic	0.849	0.685	0.382	0.569	0.738	0.527
Discriminant	0.632	0.821	0.559	0.594	0.742	0.593
Tree	0.547	0.819	0.564	0.555	0.698	0.555
RuleSet	0.500	0.836	0.630	0.561	0.680	0.557
Boosting	0.434	0.835	0.651	0.531	0.638	0.521
IBk	0.509	0.812	0.548	0.528	0.673	0.528
DecisionStump	0.642	0.808	0.529	0.583	0.739	0.580
VotedPerceptron	0.849	0.376	0.228	0.440	0.463	0.360
VF1	0.887	0.586	0.319	0.532	0.671	0.469
ROCKY	0.722	0.727	0.409	0.543	0.725	0.522
RF(cutoff=(0.9,0.1))	0.880	0.723	0.419	0.607	0.774	0.567
RF(cutoff=(0.8,0.2))	0.806	0.769	0.465	0.612	0.782	0.590
RF(cutoff=(0.7,0.3))	0.620	0.784	0.482	0.547	0.716	0.543

Table 8: Performance Comparison on Project CM1

Methods	PD(Recall)	ACC	Precision	$G - mean_1$	$G - mean_2$	F-measure
Logistic	0.776	0.657	0.192	0.386	0.707	0.308
Discriminant	0.694	0.841	0.346	0.490	0.771	0.462
Tree	0.204	0.906	0.561	0.338	0.448	0.299
RuleSet	0.102	0.882	0.253	0.161	0.314	0.145
Boosting	0.020	0.892	0.145	0.054	0.141	0.035
KStar	0.204	0.859	0.243	0.222	0.436	0.222
NaiveBayes	0.306	0.849	0.267	0.286	0.527	0.285
VF1	0.898	0.339	0.120	0.328	0.450	0.211
RF(cutoff=(0.9,0.1))	0.714	0.651	0.179	0.358	0.678	0.287
RF(cutoff=(0.8,0.2))	0.490	0.805	0.250	0.350	0.641	0.331
RF(cutoff=(0.7,0.3))	0.224	0.855	0.244	0.234	0.456	0.234

Table 9: Comparing RF and BRF by Use of AUC

Projects	AUC (RF)	AUC (BRF)
JM1	0.7471	0.7616
PC1	0.8504	0.8781
KC1	0.8309	0.8245
KC2	0.8424	0.8432
CM1	0.7514	0.7750

BRF are minimal on all the NASA project. BRF seems to be slightly better than RF in most of the cases. We wanted to further quantify this difference by measuring the area under the ROC curve (Area Under the Curve - AUC). The AUC is a measure of the overall performance of an analytical test and is interpreted as the average value of sensitivity for all possible values of specificity. It can take on any value between 0 and 1, since both the x and y axes have values ranging from 0 to 1. An AUC value of 1 indicates a perfectly accurate classifier. An AUC of 0.50 means that the accuracy is equivalent to that which would be obtained by random guessing. The area under the curve is calculated using the trapezoid rule. *Table 9* summarizes AUC information. Balanced random forests provide a better classification of fault-prone modules over the conventional random forests algorithm on projects JM1, PC1 and CM1. Of the remaining two projects, in KC1 there is a slight decrease in the value of AUC, while KC2 shows a slight improvement. However, performance differences between RF and BRF on these two data sets are not statistically significant.

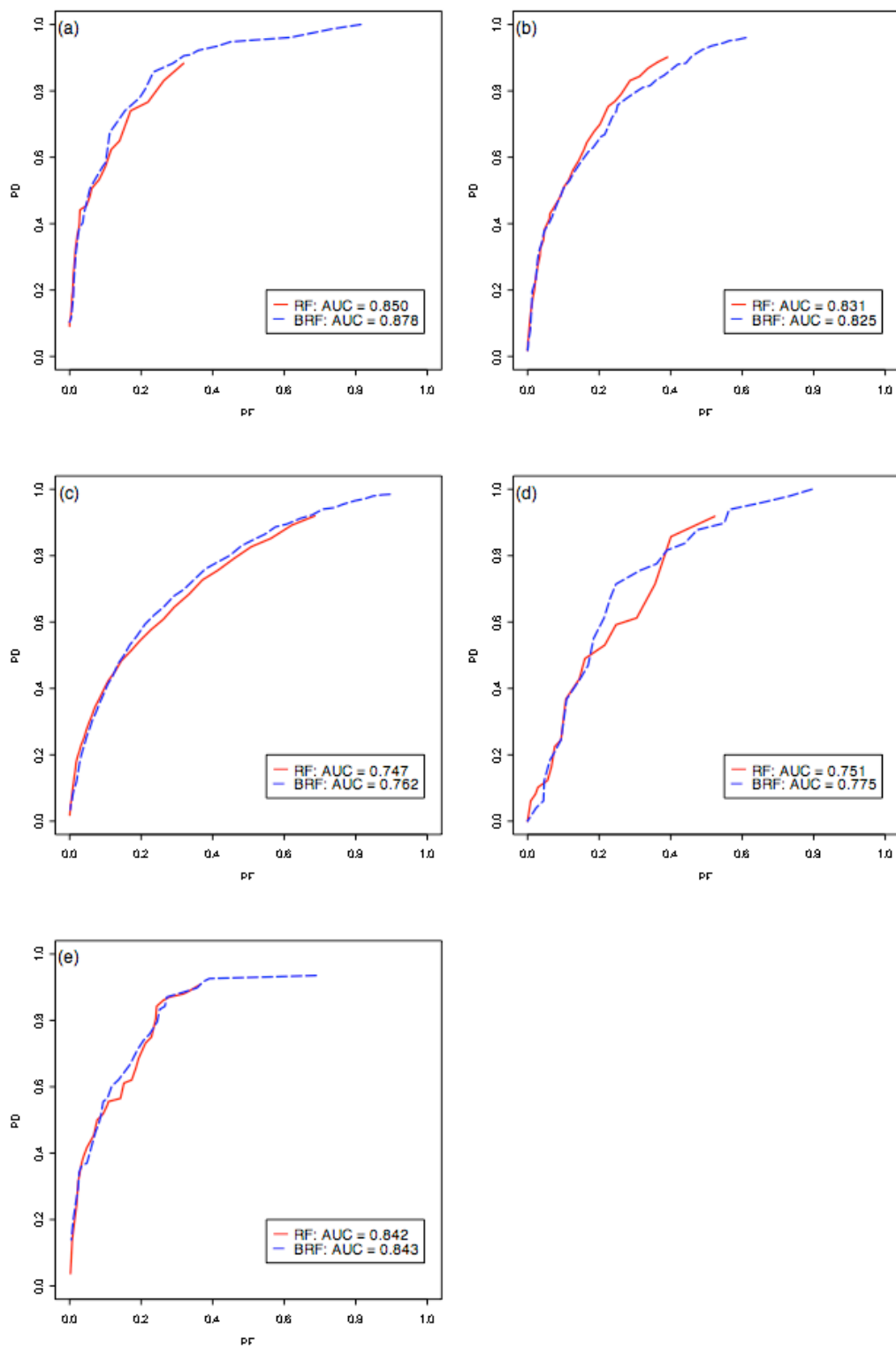


Figure 4: AUC of RF and Balanced RF Algorithms. The red line is the ROC produced in RF and the blue dashed line is generated in BRF. (a). PC1 (b). KC1 (c). JM1 (d). CM1 (e). KC2

7 Discussion

This paper compares many statistical methods and machine learning algorithms in the context of predicting fault-prone modules in software development. Our experiments demonstrate that some of the evaluated classification algorithms (for example, RuleSet, Boosting, single tree classifiers) are not likely to perform well in the context of software quality prediction, even though they may have been considered a few times in the literature over the past decade. Many classifiers used in our performance comparison exhibit either a low defect detection rate PD or a low overall accuracy, or both. Other classifiers (Logistic, Discriminant, for example) should be included in the software quality engineer’s toolbox, i.e., these algorithms are the candidates for building food models for software quality prediction.

Compared with all the classifiers we got to analyze, random forests (and balanced random forests) always achieve better prediction performance, or at least achieve the the performance that matches that of the best performing classification algorithms. Random forests work especially well on large and diverse data sets, such as JM1 and PC1. As mentioned in Section 2, the prediction performance of random forests depends on the (low) correlation between trees in the forest. Each tree is built upon a bootstrapped sample from the original software metrics data set. When the sample size is not large, there is a good chance that the same instance from the training set is used in almost all the bootstrapped samples. This causes an increase in the correlation between tree classifiers in the forest and, consequently, limits the performance of the forest.

Regarding the overall performance of random forests, two important observations emerge from our experiments. One is that the random forest algorithm is always one of the best classifiers if the user-specific voting thresholds approximate the proportion of fault-prone modules in the project’s training set. The second observation relates the classification performance of two variants of the random forest algorithm. Balanced random forests provide a moderate performance increase over the traditional random forest algorithm. Since the improvement is moderate, software quality engineers have to decide whether it is worth additional effort in the design of experiments. Unlike traditional random forests and all other classification algorithms reported in this paper, balanced random forests require some extra effort as their implementation is not immediately available from off-the-shelf software tools. However, if the software project requires that as many as possible modules to be inspected due to a dire consequence of software failures, this extra effort may be warranted.

8 Summary

This paper offers several insights into the field of software quality prediction. The first and foremost observation we want the readers to remember is that the performance of predictive models for the detection of fault-prone modules in software projects needs to be evaluated from multiple points of view. Overall prediction accuracy may be a misleading criterion as faulty modules are likely to represent a minority of the modules in the database. These training sets favor the classification of software modules in the majority class, i.e., not-fault-

prone. The probability of detection index measures how likely a fault-prone module is to be correctly classified. Further, precision index indicates how many fault-free modules were misclassified as fault-prone. In case the precision index is low, many correct modules will be exposed to rigorous software verification and validation thus increasing the cost of the process and the time needed for the project completion. Due to these observation, we propose to use a different set of performance indicators when comparing the performance of classification algorithms.

We also propose a novel methodology for software quality prediction. This methodology is based on the two variants of the random forests algorithm. It is valuable for real-world applications in software quality predictions. First, it is more robust with respect to noise in the software metrics measurements than other methods. Therefore, it works especially well for the prediction of fault-proneness in large-scale software systems. Second, the methodology based on random forests runs efficiently on large data sets and provides high level of predictive power without overfitting. Therefore, based on our results, random forests are probably the most promising “best guess” when selecting a classification algorithm to apply to a data set that describes the static characteristics (software metrics) of fault-prone modules in software engineering.

Another important characteristic of this study is that it has compared results from many different classification and machine learning algorithms on five different but highly relevant software engineering data sets. The results of our experiments provide a “proving grounds” for new methods that will be developed in the future. All our experiments can be repeated, as the tools and data sets can be easily obtained by fellow researchers. Thus, new methods for predicting fault-prone modules in software development projects can be and probably should be compared with our results in a clear and consistent way.

References

- [1] Software package *See5*, (2005). From <http://www.rulequest.com/see5-info.html>.
- [2] Azar, D., Bouktif, S., Kégl, B., Sahraoui, H. & Precup, D. (2002). Combining And Adapting Software Quality Predictive Models By Genetic Algorithms, *Proc. 17th IEEE International Conference on Automated Software Engineering (ASE2002)*, p285.
- [3] Basili, V. R., Briand, L. C. & Melo, W. (1996). A Validation of Object-oriented Design Metrics as Quality Indicators, *IEEE Trans. Software Eng.*, 22(10): 751-761.
- [4] Boetticher, Gary D. (2004). Nearest Neighbor Sampling for Better Defect Prediction, Retrieved June 2005, from <http://promise.site.uottawa.ca/proceedings/pdf/7.pdf>.
- [5] Breiman, L. (2001). Random Forests, *Machine Learning*, vol. 45, 5-32.
- [6] Breiman, L. & Cutler, A. (2004). Random Forests: Classification/Clustering, Retrieved May 2004, from <http://www.stat.berkeley.edu/users/breiman/RandomForests>.

- [7] Breiman, L. Wald Lecture II, Looking Inside the Black Box, Retrieved May 2004, from <http://www.stat.berkeley.edu/users/breiman>.
- [8] Breiman, L, Friedman, J. H., Olshen, R. A. & Stone, C. J. (1984). *Classification and Regression Trees*, Wadsworth.
- [9] Breiman, L. (1996). Bagging Predictors, *Machine Learning*, vol. 24, 123-140.
- [10] Briand, L. C., Basili, V. R. & Hetmanski, C. J. (1993). Developing Interpretable Models with Optimaized Set Reduction for Identifying High-Risk Software Components, *IEEE Trans. Software Eng.*, 19(11): 1028-1044.
- [11] Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). Smote: Synthetic Minority Over-sampling Technique, *Journal of Artificial Intelligence Research*, 16, 321-357.
- [12] Chen, C., Liaw, A. & Breiman, L. (2004). Using Random Forest to Learn Imbalanced Data, Retrieved October 2004, from <http://stat-www.berkeley.edu/users/chenchao/666.pdf>.
- [13] Dettling, M. (2004). BagBoosting for Tumor Classification with Gene Expression Data. *Bioinformatics*, vol. 20, No. 18.
- [14] Ebert, C. & Baisch, E. (1998). Industrial Application of Criticality Prediction in Software Development, *Proc. of the Nineth Iternational Symposium on Software Reliability Eng.*, p80.
- [15] Ebert, C. (1996). Classification Techniques for Metric-based Software Development, *Software Quality Journal*, 5(4): 255-272.
- [16] Fenton, N. & Neil, M. (1999). Software Metrics and Risk, *Proc. 2nd European Software Measurement Conference*, TI-KVIV, Amsterdam.
- [17] Freund, Y. & Schapire, R. E. (1998). Large margin classification using the perceptron algorithm, *Proc. 11th Annu. Conf. on Comput. Learning Theory*, 209-217, ACM Press, New York, NY.
- [18] Gokhale, S. S. & Lyu, M. R. (1997). Regression tree modeling for the prediction of software quality, *Proc. the Third ISSAT International Conference on Reliability and Quality in Design*, pp31-36, Anaheim, CA.
- [19] Guo, L., Cukic, B. & Singh, H. (2003). Predicting Fault Prone Modules by the Dempster-Shafer Belief Networks, *Proc. 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*.
- [20] Hanley, J. A., & McNeil, B. J. (1983). A Method of Comparing the Areas under Receiver Operating Characteristic Curves Derived from the Same Cases, *Radiology*, 148, 839-843.

- [21] Hanley, J. A. & McNeil B. J. (1982). The Meaning and Use of the Area under a Receiver Operating Characteristic (ROC) Curve, *Radiology*, 143, 29-36.
- [22] Hastie, T., Tibshirani, R., & Friedman, J. (2001). *The Elements of Statistical Learning*, Springer Series in Statistics, Springer-Verlag, New York.
- [23] Hudepohl, J., Aud, S. J., Khoshgoftaar, T. M., Allen, E. B. & Maryland, J. (1996). Emerald: Software Metrics and Models on the Desktop, *IEEE Software*, pp. 56-60.
- [24] John, G. Cleary & Leonard, E. Trigg (1995). K*: An Instance-based Learner Using an Entropic Distance Measure, *Proc. of the 12th International Conference on Machine Learning*, 108-114.
- [25] Khoshgoftaar, T. M., Allen, E. B., Ross, F. D., Munik oti, R., Goel, N. & Nandi, A. (1997). Predicting Fault-Prone Modules with Case-Based Reasoning, *Proc. the Eighth International Symposium on Software Engineering (ISSRE'97)*, pp27.
- [26] Khoshgoftaar, T. M., Allen, E. B., Kalaichelvan, K. S. & Goel, N. (1996). Early Quality Prediction: A Case Study in Telecommunications, *IEEE Software*, 13(1): 65-71.
- [27] Khoshgoftaar, T. M. & Lanning, D. L. (1995). A Neural Network Approach for Early Detection of Program Modules Having High Risk in the Maintenance Phase, *Journal of Systems and Software*, 29(1): 85-91.
- [28] Khoshgoftaar, T. M. & Seliya, N. (2002). Tree-Based Software Quality Estimation Models For Fault Prediction, *Proc. the Eighth IEEE Symposium on Software Metrics (METRICS'02)*, pp203.
- [29] Kubat, M., Holte, R., & Matwin, S. (1998). Machine Learning for the Detection of Oil Spills in Satellite Radar Images, *Machine Learning*, vol. 30, 195-215.
- [30] Lewis, D. & Gale, W. (1994). A Sequential Algorithm for Training Text Classifiers, *Proc. of Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, Springer-Verlag, London, 3-12.
- [31] Menzies, T. Stefano, Ammar, J. D., Chapman, K., McGill, R. M., Callis, K. & Davis, J. (2003). When Can We Test Less? *IEEE Metrics 2003*, Available at: <http://menzies.us/pdf/03metrics.pdf>.
- [32] Munson, J. C. & Khoshgoftaar, T. M. (1992). The Detection of Fault-prone Programs, *IEEE Trans. Software Eng.*, 18(5): 423-433.
- [33] Quinlan, J. R. (1993). *C4.5 Programs for Machine Learning*, Morgan Kaufmann.
- [34] Remlinger, K. S. (2003). Introduction and Application of Random Forest on High Throughput Screening Data from Drug Discovery, Retrieved October 2003, from <http://www4.ncsu.edu/~ksremlin>.

- [35] Schneidewind, N. F. (1992). Methodology For Validating Software Metrics, *IEEE Trans. Software Eng.*, 18(5): 410-422.
- [36] Selby, R. W. & Porter, A. A. (1988). Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis, *IEEE Trans. Software Eng.*, 14(12): 1743-1756.
- [37] Shawe-Taylor, J. & Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*, Cambridge, UK: Cambridge University Press.
- [38] Speed, T. (Ed.). (2003). *Statistical Analysis of Gene Expression Microarray Data*, Chapman & Hall/CRC Press LLC.
- [39] Troster, J. & Tian, J. (1995). Measurement and Defect Modeling for a Legacy Software System, *Annals of Software Eng.*, 1: 95-118.
- [40] Witten, I. H. & Frank, E. (1999). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann.