Misclassification cost-sensitive fault prediction models

Yue Jiang, Bojan Cukic, Tim Menzies
The Lane Department of Computer Science and Electrical Engineering
West Virginia University
Morgantown, WV26506-6109
{yue, cukic, menzies}@csee.wvu.edu

ABSTRACT

Traditionally, software quality models are built by assuming a uniform misclassification cost. In other words, cost implications of misclassifying a fault prone module as fault free are assumed to be the same as the cost implications of misclassifying a fault free module as fault prone. In reality, these two types of misclassification costs are rarely equal. They are project-specific, reflecting the characteristics of the domain in which the program operates. In this paper, through the analysis of projects from a public repository, we analyze the benefits of techniques which incorporate misclassification costs in the development of software quality models. We find that cost-sensitive learning does not provide operational points which outperform cost-insensitive classifiers. However, an advantage of cost-sensitive modeling is the explicit choice of the operational threshold appropriate for the cost differential.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: metrics—*quality, performance*

General Terms

Design, Experimentation, Performance

Keywords

Fault-proneness prediction, Machine learning, Cost-sensitive

1. INTRODUCTION

1.1 Fault Prediction Models

Research studying the detection of software modules which are likely to contain faults has been ongoing for a long time. The fault-proneness information not only points to the need for increased quality monitoring during the development but also provides important advice to assign verification and validation activities. Various studies show that software companies spend 50% to 80% of their software development efforts on testing [13]. The identification of fault-prone modules might have a significant cost-saving impact on software development.

A wide range of software metrics have been proposed for collection and used to identify modules which may contain faults [12]. Metrics describing software requirements (i.e. requirement documents) achieved notable success in predicting fault prone modules [19, 27, 16]. Design metrics, either collected from design documents or reverse engineered from code, have proved their utility for fault-proneness prediction [32, 34]. Static code metrics, such as Halstead complexity [15] and various code size metrics, have also proven their effectiveness in many studies [28, 31, 22, 14, 6].

1.2 Importance of Cost

The "traditional" software quality models, some referenced above and others overviewed in Section 5, typically assume uniform misclassification cost. In other words, this models suppose that the cost implications of wrongly predicting a faulty module as fault free one is the same as the cost of indicating that a fault free module may contain faults. In reality, the cost implications of these two types of misclassification are seldom equal in the real world. In high risk software projects, for example, safety-related spacecraft navigation instruments, the cost of missing a faulty module may have extreme consequence associated with a loss of the entire mission. Therefore, in such projects significant resources are typically available for identifying and eradicating all faults because the cost of losing a mission is much higher. On the other hand, in low risk projects which aim to occupy a new market niche, time to market pressure may imply that only a minimal number of faulty modules can be analyzed. The cost of analyzing any significant number of misclassified fault free modules is, therefore, unacceptable.

What makes the prediction of faulty modules a challenge is the reality that they usually form a minority class compared to fault free modules. The faulty and non-faulty classes are typically imbalanced. Therefore, in order to develop models which find more faults, one would expect that explicit placement of a cost premium for fault identification will increase the accuracy of such models. This expectation represents the motivation for the research described in this paper.

1.3 Cost in Model Evaluation

While the techniques for model development which explicitly account for misclassification cost differential have not been studied in software quality modeling, there have been attempts to include cost factors into model evaluation. *F-measure*, for example, offers a technique to account for the cost factor [18] when comparing different models. Khoshgoftaar and Allen [23] proposed the use of prior probabilities of misclassification to select classifiers which offer the most appropriate performance. In [24] they compare the return-on-investment in a large legacy telecommunication system when V&V activities are applied to software modules selected by a

	actual faulty	actual non-faulty
predict faulty	TP	FP
predict non-faulty	FN	TN

$$\begin{aligned}
yRecall = yPD = &\ \frac{TP}{TP+FN} \\
yPF = &\ \frac{FP}{FP+TN} \\
yPrecision = &\ \frac{TP}{FP+TP} \\
nRecall == nPD = &\ \frac{TN}{FP+TN} \\
nPF = &\ \frac{FN}{TP+FN} \\
nPrecision = &\ \frac{TN}{TN+FN}
\end{aligned}$$

Figure 1: Confusion Matrix

quality model vs. at random. Cost has been considered in test case selection for regression testing too [11].

There is a steady trend in fault prediction modeling literature recommending model evaluation with lift charts [18], sometimes called Alberg diagrams [32, 33]. Lift is a measure of the effectiveness of a classifier in the detection faulty modules. It calculates the ratio of correctly identified faulty modules with and without the predictive model. Lift chart is especially useful when the project has resources to apply verification activities to a limited number of modules. Cost effectiveness measure described by Arisholm *et al.* [2] can account for the nonuniform cost of module-level $V\&V$. As opposed to these approaches, our goal is to analyze the benefits of incorporating project specific misclassification costs in the development of software quality models, as opposed to their evaluation.

The remainder of this paper is organized as follows. Section 2 introduces our method of cost-sensitive modeling and our experimental design. Section 3 presents our experimental result and analysis. Section 4 offers a short overview of related work. Section 5 concludes the paper.

2. COST-SENSITIVE MODELING

In spite of the importance of misclassification cost in software quality modeling, most classifiers simply do not allow the incorporation of cost into the modeling process. Instead they are typically designed to increase the overall prediction accuracy (or decrease the overall error rate) assuming that all misclassifications have the same cost.

In this section, we introduce a method to incorporate different misclassification costs into software quality modeling. In these cost-sensitive models, the goal will be to minimize the overall misclassification cost. This is quite different from the cost-insensitive models. First, we discuss the confusion matrix and the corresponding measurements used in this study. Then we explain our cost-sensitive classification methods. Finally, we explain experimental design used to analyze the results.

2.1 Confusion Matrix

Figure 1 shows a confusion matrix. It has four categories: True positives (TP) are modules correctly classified as faulty modules. False positives (FP) refer to fault-free modules incorrectly labeled as faulty. True negatives (TN) correspond to correctly classified fault-free modules. Finally, false negatives (FN) refer to faulty modules incorrectly classified as fault-free. In this study, we will use $recall, precision$, and PF to evaluate prediction models. $Recall$ represents the probability of detection (PD) of faulty (or non-faulty) modules. In this study, $recall$ of faulty modules is denoted as $yRecall$, $recall$ of non-faulty modules is $nRecall$. $Precision$ is the proportion of the correctly predicted (faulty or non-faulty) modules inside each prediction class: precision for faulty modules is denoted $yPrecision$, for non-faulty modules $nPrecision$. PF represents the probability of false alarms: PF for faulty modules is denoted as yPF, for non-faulty modules nPF.

	actual faulty	actual non-faulty
predict faulty	0 (TP)	1(FP)
predict non-faulty	5(FN)	0(TN)

Figure 2: Cost matrix, an example

Receiver Operating Characteristic (ROC) curve is a plot of the probability of detection ($recall$ or PD) as a function of the probability of false alarm (PF) across all threshold settings. An ROC curve provides an intuitive way to evaluate the classification performance of software quality models. Many classifiers allow users to define and adjust threshold parameters in order to generate an appropriate performance [35]. The Area Under the ROC Curve (AUC) is a numeric performance evaluation measure directly associated with an ROC curve. In this study, we will utilize ROC and AUC for model evaluation.

Boxplot diagrams, also known as box and whisker diagrams, graphically depict numerical data distributions using the five first order statistics: the smallest observation, lower quartile(Q1), median, upper quartile(Q3), and the largest observation. The box is constructed based on the interquartile range(IQR) from Q1 to Q3. The line inside the box depicts the median which follows the central tendency. Outliers may be indicated as bubbles (or squares) lying below/above 1.5*IQR. The whiskers indicate the smallest and the largest observations which are not outliers.

2.2 Modeling Algorithms

In this study, the term cost always stands for the $misclassification\ cost$. Figure 2 shows a cost matrix similar to the confusion matrix shown in Figure 1. The cost matrix emphasizes the implications of misclassification. For this reason, the costs of correct classifications are 0. If a fault free module is misclassified as faulty (FP), additional $V\&V$ activity imply additional expenditure. In Figure 1, this misclassification cost is assumed to be a factor 1. If a faulty module is misclassified as fault free (FN), the cost indicates the potential for future damages. In Figure 1, it is assumed to be 5 times more expensive than FP.

The MetaCost method introduced by Domingos [9] makes classifiers cost-sensitive. Let i, j denote two classes, m denotes a module. The probability $P(j|m)$ stands for the probability that m belongs to class j. $C(i,j)$ denotes the costs denoted in the cost matrix. MetaCost tries to minimize the misclassification cost by using Bayes optimal prediction:

$$R(i|m) = \sum_j P(j|m)C(i,j) \qquad (1)$$

The misclassification cost $R(i|m)$ is the expected cost of predicting that module m belongs to class i. MetaCost aims to achieve the lowest possible overall misclassification cost over all modules by wrapping a procedure described in Equation 1 to a regular cost-insensitive classifier [9]. MetaCost works as follows [9]:

- First, bootstrap the training subset to form a sample; run the base cost-insensitive classifier on the bootstrapped sample. This procedure is repeated multiple times.

- Estimate class' probability $P(j|m)$ for each module m, across all bootstraped samples.
- Given the known cost matrix, $C(i,j)$, using Equation 1 relabel each module with the estimated optimal class in order to minimize the overall misclassification cost, $R(i|m)$. This process results in an optimal model.
- Apply the model developed accordingly to the test subset samples.

Observe that with different cost matrices, the last two steps need to be adjusted accordingly. According to Domingos, "MetaCost treats the underlying classifier as a blackbox, requiring no knowledge of its functioning or change to it" [9]. Weka offers MetaCost procedure proposed by Domingos as a wrapper to any supported classifier.

2.3 Design of Experiments

The *13* data sets listed in Table 1 come from the NASA Metrics Data Program (MDP) repository [1]. The projects offer module metrics associated with NASA Space Shuttle software. JM1 and KC1 have 21 attributes that can be used as predictor variables, MC1 and PC5 have 39, while the other data sets have 40.

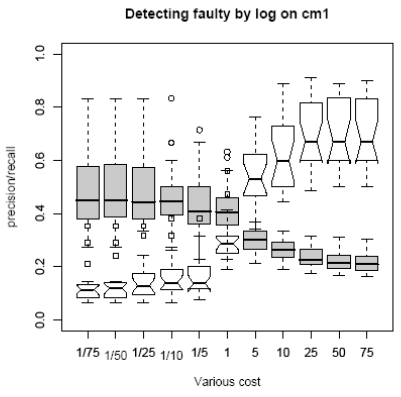
We do not know the cost matrix appropriate for each project. In order to investigate the effect of cost-sensitive classification in software quality models, we assign 11 different cost matrices to all projects shown in Table 2. In Table 2, the cost of TP and TN are zero. We use $cost\_ratio$ to denote a cost matrix which is shown in Table 2 as the third column. For example, in the first row, the ratio between FN and FP is $\frac{1}{75}$. The first five cost matrices represent for low risk projects and the last five cost matrices stand for the high risk projects.

The 11 cost ratios from Table 2 are run on each classifier over each data set. We use standard 10 way cross-validation (CV) method in this study. CV is the statistical practice of randomized partitioning of a data set into two parts: one part of data is used as training and the remaining part of data is left as testing subset. In this study, 50% data is used for training and the other 50% is used for testing. The partition is randomized 10 times and run 20 times on each data set to get a better understanding of variance.

In experiments, we use five classifiers from Weka which consistently demonstrate good performance [18, 20, 17, 19]: random forest(rf), boosting(bst), logistic(log), naiveBayes(nb), and bagging(bag). The measures evaluated in this study are $precision$ ($yPrecision$ and $nPrecision$) and $recall$ ($yRecall$ and $nRecall$). Using 13 data sets, 11 different cost matrices, 5 machine learners, and 20 cross validation runs, in total, our experiments resulted in $14,300$ runs.

3. ANALYSIS OF EXPERIMENTS

For each classifier – data set combination, we generate two boxplot diagrams, which depict $precision$ and $recall$ for faulty and for non-faulty modules. In total, there are $13 * 5 * 2 = 130$ such diagrams. Understandably, we cannot describe all our observations in a paper, and neither would it be very interesting. However, we observed the following trends: logistic and boosting classifiers offer very similar performance characteristics; bagging is similar to random forests; NaiveBayes is quite different as the impact of cost, measured by $precision$ or $recall$, seems to have a minor, if any, impact. Therefore, we use logistic and bagging classifiers as examples to illustrate observed results. Figure 3 depicts boxplot diagrams for $precision$ and $recall$ of logistic and bagging on CM1 and JM1 data sets.



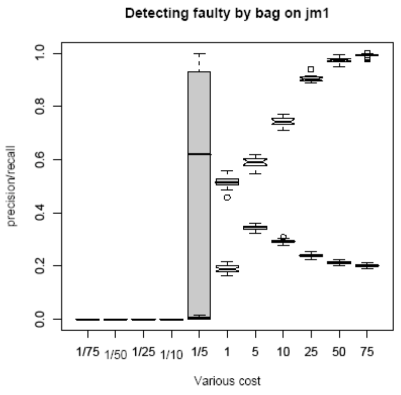


Figure 3: Boxplot diagrams depict $precision$ and $recall$ for identifying faulty and non-faulty modules using different $cost\_ratio$. Shaded rectangle stands for $precision$; unshaded notch stands for $recall$. Outliers for $precision$ are indicated as bubbles; and outliers for $recall$ are indicated as squares.

From Figure 3, we can observe the clear increase of $recall$ with the decrease of $precision$ in the detection of faulty modules when $cost\_ratio$ varies from $1/75$ to 75. However, the rate of increase depends on the classifier. As expected, when identifying fault free modules, $recall$ decreases as $precision$ increases.

Table 3 shows the median $precision$, $recall$, and PF for faulty and fault free classes for 5 classifiers on CM1 and JM1 two data sets. The performance indices for $cost\_ratio$ of $1/25$, $1/5$, 5, and 25 are omitted as they follow the overall performance trends. In low risk situations ($cost\_ratio < 1$) misclassifying fault free modules is more expensive than misclassifying faulty ones. $yRecall$ is, therefore, very low. For example, in CM1 project from 0.25 to 0.26 using naive bayes down to 0 with bagging and random forest. $yPrecision$ is not good either. On the contrary, $nRecall$ and $nPrecision$ are all quite high, greater than 0.80 with some even as high as 1. This is not difficult to explain for field use data. When there are limited resources to verify modules in a software project, the cheapest choice is to check as few modules as possible. The behavior of bagging and random forest in these circumstances mimics trivial classifiers by classifying every module as fault free.

In high risk projects ($cost\_ratio > 1$) misclassifying a faulty module as fault free is undesirable. The data sets used in this study fall into this category, i.e., in most cases the models should identify as many faults as possible. The good news is that when we increase

Table 1: Datasets used in this study

Data	mod.#	% faulty	project description	lang.
JM1	10,878	19.3%	Real time predictive ground system	C
MC1	9466	0.64%	Combustion experiment of a space shuttle	(C)C++
PC2	5589	0.42%	Dynamic simulator for attitude control systems	C
PC5	17,186	3.00%	Safety enhancement system	C++
PC1	1109	6.59%	Flight software from an earth orbiting satellite	C
PC3	1563	10.43%	Flight software for earth orbiting satellite	C
PC4	1458	12.24%	Flight software for earth orbiting satellite	C
CM1	505	16.04%	Spacecraft instrument	C
MW1	403	6.7%	Zero gravity experiment related to combustion	C
KC1	2109	13.9%	Storage management for ground data	C++
KC3	458	6.3%	Storage management for ground data	Java
KC4	125	48%	Ground-based subscription server	Perl
MC2	161	32.30%	Video guidance system	C++

Table 2: Eleven different cost matrices assigned to 13 projects in this experiment.

cost of FN	cost of FP	denoted as cost_ratio	risk type	note
1	75	1/75	low	lower cost to misclassifying faulty modules
1	50	1/50	low	
1	25	1/25	low	
1	10	1/10	low	
1	5	1/5	low	
1	1	1		equal cost to both classes
5	1	5	high	higher cost to misclassifying faulty modules
10	1	10	high	
25	1	25	high	
50	1	50	high	
75	1	75	high	

$cost\_ratio$ from 1 to 75, $yRecall$ increases, although the rate of increase depends on the algorithm. The bad news, although not unexpected, is the decrease of $yPrecision$ which implies needless analysis of a large number fault free modules (false positives). The optimal goal is to have high $recall$ and $precision$ at the same time, but that seems impossible to achieve by varying $cost\_ratio$ only. For example, in JM1 project when $cost\_ratio$ is greater than 50, $yRecall$ increases to 1 (using logistic and boosting) while $nRecall$ decreases to 0. This, again, reflects the result of a trivial classifier which tags every module as faulty.

Figure 3 and Table 3 lead us towards the following observations:

- Fault prone modules are the minority class. Different cost parameters indicate a compromise between $yPrecision$ and $yRecall$. Nevertheless, some classifiers offer better tradeoffs between the two evaluation parameters.
- Boosting, bagging and random forest algorithms consistently reach $yRecal$ rates close to 1 at high cost ratios, with precision slightly above 0.2.
- No matter what the $cost\_ratio$ is, $nPrecision$ and $nRecall$ for identifying fault free modules are very high, reflecting the fact that this is the majority class.
- NaiveBayes is quite different from the other four classifiers. It's performance indices ($precision$,$recall$, PF) are rather constant regardless of the cost.

We also want to acknowledge the high variance of $yPrecision$ in Figure 3 at $cost\_ratio = 1/5$. Essentially, the precision in identifying faulty modules varies from 0 to close to 1. We noticed similar spikes in precision variance in a few other experiments. For example, random forest on JM1 project at $cost\_ratio = 1/25$, boosting on KC3 data with $cost\_ratio$ between $1/75$ and $1/5$, bagging on PC1 data with $cost\_ratio = 1$ all report very high variance for precision. Similarly, when the goal is to identify fault free modules, $nPrecision$ at high $cost\_ratio$ may have a big variance too: NaiveBayes on MC1 with $cost\_ratio$=75, random forest on MC2 at $cost\_ratio$ 50 and 75, logistic with JM1 with $cost\_ratio$ between 50 and 75. These observations point that future research in software quality modeling must take such variances into account when recommending best practices. With such unreliable performance, it is difficult to trust prediction results. The good news is that recall does not appear to suffer from big variances through all the experiments in this study. Therefore, $recall$ should be relied upon when evaluating the performance of software quality models.

3.1 Statistical significance

We further conducted a statistical test procedure to compare $yRecall$ across 11 different cost matrices for each classifier with each data set, using Demsar's test [7, 18]. Our hypotheses are: H_0: *There is no difference in the performance among 11 different cost matrices for a classifier on a specific data set evaluated using* $yRecall$.
vs.
H_α: *At least two different cost matrices have significantly different performance for a learner on a specific data set evaluated by using* $yRecall$.

Using 95% confidence interval to evaluate the significance of test, five classifiers and 13 data sets, we conducted Demsar's test 65 times. Amongst the 65 tests, only two accepted H_0 and rejected H_α: naive bayes on JM1 and MW1. In other 63 tests, the hypothesis H_0 is rejected and H_α is accepted. The $yRecall$ based performance ranks prefer higher $cost_ratio$s.

Statistically significant differences in performance when all 11 cost ratios are taken into account are not surprising. Another important question is how sensitive models are to small changes in

Table 3: Median of $precision$ and $recall$ for 5 classifiers on CM1 and JM1 data set. $yPrecision$ and $yRecall$ stand for $precision$and $recall$ for faulty modules; $nPrecision$ and $nRecall$ are corresponding $precision$ and $recall$ rates for identification of fault free modules.

dataset	learner	cost_ratio	yPF	yPrecision	yRecall	nPF	nPrecision	nRecall
	log	1/75	0.03	0.45	0.11	0.89	0.85	0.97
	log	1/50	0.03	0.45	0.12	0.88	0.85	0.97
	log	1/10	0.03	0.45	0.14	0.86	0.85	0.97
	log	1	0.08	0.41	0.29	0.71	0.87	0.92
	log	10	0.34	0.26	0.6	0.4	0.9	0.66
cm1	log	50	0.48	0.21	0.67	0.33	0.9	0.52
	log	75	0.49	0.21	0.67	0.33	0.9	0.51
	bst	1/75	0	0.54	0.05	0.95	0.84	1
	bst	1/50	0.01	0.59	0.04	0.96	0.84	1
	bst	1/10	0.01	0.41	0.03	0.98	0.84	0.99
	bst	1	0.03	0.43	0.13	0.87	0.85	0.97
	bst	10	0.38	0.25	0.7	0.3	0.91	0.62
cm1	bst	50	0.63	0.21	0.88	0.13	0.94	0.37
	bst	75	0.64	0.22	0.9	0.1	0.95	0.36
	bag	1/75	0	0	0	1	0.84	1
	bag	1/50	0	0	0	1	0.84	1
	bag	1/10	0	0	0	1	0.84	1
	bag	1	0.02	0.35	0.06	0.94	0.84	0.98
	bag	10	0.33	0.26	0.63	0.37	0.91	0.67
cm1	bag	50	0.71	0.2	0.93	0.07	0.96	0.29
	bag	75	0.79	0.19	0.98	0.02	0.98	0.21
	rf	1/75	0	0	0	1	0.84	1
	rf	1/50	0	0	0	1	0.84	1
	rf	1/10	0	0	0	1	0.84	1
	rf	1	0.03	0.55	0.19	0.81	0.86	0.97
	rf	10	0.17	0.33	0.47	0.54	0.89	0.83
cm1	rf	50	0.53	0.23	0.84	0.16	0.94	0.47
	rf	75	0.62	0.22	0.94	0.06	0.97	0.38
	nb	1/75	0.08	0.36	0.25	0.75	0.86	0.92
	nb	1/50	0.08	0.36	0.25	0.75	0.86	0.92
	nb	1/10	0.08	0.36	0.26	0.74	0.87	0.92
	nb	1	0.09	0.34	0.27	0.73	0.87	0.91
	nb	10	0.1	0.33	0.29	0.71	0.87	0.9
cm1	nb	50	0.11	0.32	0.3	0.7	0.88	0.89
	nb	75	0.11	0.32	0.3	0.7	0.88	0.89
	log	1/75	0	0.77	0.01	0.99	0.81	1
	log	1/50	0	0.76	0.01	0.99	0.81	1
	log	1/10	0	0.81	0.01	0.99	0.81	1
	log	1	0.02	0.6	0.12	0.88	0.82	0.98
	log	10	0.83	0.22	0.94	0.06	0.93	0.17
jm1	log	50	1	0.19	1	0	0.54	0
	log	75	1	0.19	1	0	0	0
	bst	1/75	0	0.88	0.01	0.99	0.81	1
	bst	1/50	0	0.89	0.01	0.99	0.81	1
	bst	1/10	0	0.85	0.01	0.99	0.81	1
	bst	1	0.03	0.52	0.11	0.89	0.82	0.97
	bst	10	0.76	0.23	0.94	0.06	0.94	0.24
jm1	bst	50	1	0.19	1	0	0	0
	bst	75	1	0.19	1	0	0	0
	bag	1/75	0	0	0	1	0.81	1
	bag	1/50	0	0	0	1	0.81	1
	bag	1/10	0	0	0	1	0.81	1
	bag	1	0.04	0.51	0.19	0.81	0.83	0.96
	bag	10	0.43	0.29	0.74	0.26	0.9	0.57
jm1	bag	50	0.87	0.21	0.98	0.03	0.96	0.13
	bag	75	0.95	0.2	0.99	0.01	0.97	0.06
	rf	1/75	0	0	0	1	0.81	1
	rf	1/50	0	0	0	1	0.81	1
	rf	1/10	0	0.87	0.01	1	0.81	1
	rf	1	0.05	0.53	0.22	0.78	0.84	0.95
	rf	10	0.24	0.36	0.57	0.44	0.88	0.76
jm1	rf	50	0.69	0.24	0.91	0.09	0.94	0.31
	rf	75	0.8	0.22	0.95	0.05	0.95	0.2
	nb	1/75	0.05	0.49	0.18	0.82	0.83	0.95
	nb	1/50	0.05	0.49	0.18	0.82	0.83	0.95
	nb	1/10	0.05	0.49	0.19	0.81	0.83	0.95
	nb	1	0.05	0.48	0.2	0.8	0.83	0.95
	nb	10	0.06	0.48	0.21	0.79	0.83	0.94
jm1	nb	50	0.06	0.48	0.22	0.78	0.83	0.94
	nb	75	0.06	0.47	0.22	0.78	0.84	0.94

the misclassification cost ratio. For this test, we will apply Demsar's procedure to compare the rank performance of classification models for each classifier using costs which range from 1 to 75.

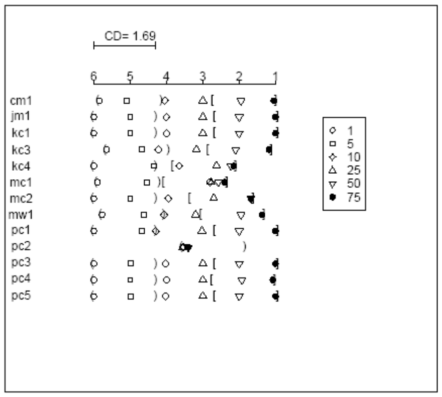


Figure 4: Demsar's rank test states that the rank performance of classifiers within the critical distance ($CD = 1.65$) is not statistically different. Classifiers were developed using random forest algorithm and misclassification costs from 1 to 75.

Figure 4 depicts the results for random forest classifier over all the 13 projects. Any two classifiers which lay within the critical distance ($CD = 1.69$) offer performance which cannot be interpreted as different using 95% confidence interval. Using $yRecall$ as the performance measure, in most data sets models developed using cost ratios 50 and 75, or 1 and 5 perform similarly. Analysis of other classifiers offer similar results. For this reason, we conclude that knowing the exact misclassification cost is not as important as knowing its range range.

Using $recall$ to evaluate the models' performance to detect faulty modules, we conclude that fault prediction generally benefits from cost-sensitive learning in high risk cases because higher $recall$ values, which indicate higher performance ranks, are obtained by applying higher. But the bottom line question is whether similar performance could have been achieved through the traditional model development, not "burdened" by the various misclassification cost factors.

3.2 Cost-sensitive vs. cost-ignorant

Figure 5 shows the ROC curve obtained from the boosting classifier using the traditional cost-ignorant modeling approach on CM1 data set. In the same figure, we overlayed the five median $(PF, recall)$ points of cost-sensitive learning with the values of $cost\_ratio > 1$ over the ROC curve. The cost sensitive models have been developed using the same boosting classifier wrapped in the MetaCost procedure [9]. We can observe that all five classifiers obtained through cost-sensitive modeling can be obtained from cost-ignorant model development by adjusting the model thresholds. The observations from this diagram are repeated for other classifiers and other data sets. Another example is shown in Figure 6.

To generalize, appropriate threshold selection in cost-ignorant models offers the models with performance equivalent to models derived using cost-sensitive modeling methodology. Cost-sensitive

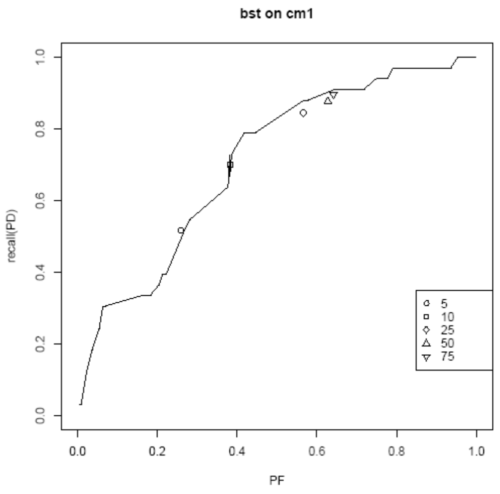


Figure 5: The ROC curve of cost-ignorant model and the median (PF, recall) points for five different $cost\_ratio > 1$ on CM1. All models developed using boosting.

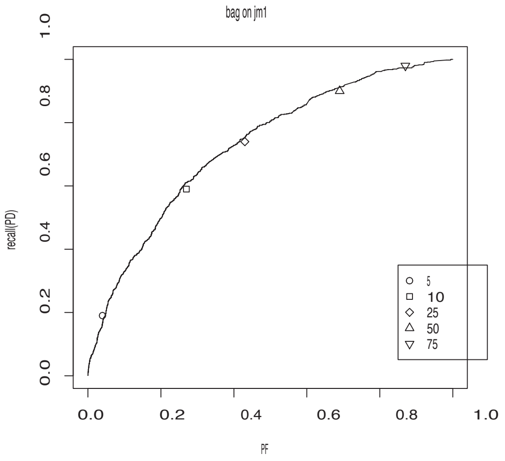


Figure 6: The ROC curve of cost-ignorant model and the median (PF, recall) points for five different $cost\_ratio > 1$. All models developed using bagging.

modeling does not provide operational points which outperform "traditional" classifiers, as evaluated $precision$ and $recall$.

The other important implication from this study is that we can use cost to choose suitable operational threshold (based on different $cost\_ratio$) to control a classifier's performance. In this study, four classifiers except naive bayes provide this flexibility.

3.3 Discussion

The experiments presented above provide valuable information in the quest to understand the best modeling practices for software quality prediction. The inclusion of misclassification cost ratio seems a natural choice in the software engineering domain, as in many cases the consequences of system failures far outweigh the cost of module verification activities.

The fact that the models developed using cost-sensitive modeling algorithm do not outperform the "traditional", cost-ignorant models has a two fold interpretation. The bad news is that this approach is not opening a new frontier in model performance. There is a good news, though. Cost-sensitive modeling explicitly reveals the model which minimizes the overall cost of software quality, given that one can trust the assumed misclassification cost ratio. Selecting a appropriate model for predicting fault-prone modules through the explicit notion of cost seems much easier for practitioners than tinkering and / or justifying one amongst many possible threshold values of the model.

In practice, exact costs are rarely known and could change as we learn more about system requirements, its design, operational environment, etc. When considering a wide range of cost ratios the resulting models differ significantly. Nevertheless, our tests indicate that changing the misclassification cost ratio from, say, 50 to 75 or 25 to 50 result in models whose performance is not significantly different. Software project managers who are likely to desire analyzing various cost situations do not need to analyze many of them, as the trends will be easy to understand.

4. RELATED WORK

A large number of fault-proneness modeling techniques have been proposed and applied to software quality prediction. Many of these techniques are relatively straightforward transplants from the fields of data mining and machine learning. Some of them, for example, logistic regression [3], aim to use domain-specific knowledge to establish the input (software metrics) output (software fault-proneness) relationship. Some techniques, such as classification trees [14, 5], neural networks [4], and genetic algorithms [21], try to examine the available large-size data sets to recognize patterns and form generalizations. Some utilize one single model to predict fault-proneness; others are based on ensemble learning and build a collection of models from a single training data set [26, 5]. Regression analysis (linear or logistic) [29, 31, 30] has the advantage that it can be used by itself but also in combinations with other algorithms,for example, classification trees, to form regression tree algorithm.

A decision tree algorithm, for example C4.5, is one of the most well studied classifiers which can depict the structure of software metrics. C4.5 uses a divide and conquer mechanism to build a decision tree from training set. After building and pruning the decision tree is fitted to the training data set. Models from decision trees are easy to interpret and, therefore, popular in practice when a fault-prone prediction model needs explanation. Khoshgoftaar *et al.* used C4.5 to identify fault-prone modules and compared it with other algorithms [25] NaiveBayes (nb), as its name suggests, "naively" assumes independence between different prediction variables. This assumption is considered overly simplistic in real life application scenarios. However, in software engineering data sets, it's performance is surprisingly good [28]. Naive Bayes classifiers have been used extensively in fault-proneness prediction, for example in [28, 19, 18].

Random Forest is a decision tree based classifier. As implied from its name, it builds a "forest" of decision trees. In empirical studies, Random forest usually is one of the best classifiers in software engineering domain [14, 19, 18]. Bagging stands for bootstrap aggregating. It relies on an ensemble of different models. The training data is resampled from the original data set. Bagging typically performs better than any single method models and almost never significantly worse. It has shown to have good performance in software engineering experiments [19, 18]. Boosting combines multiple models by explicitly seeking models that complement one another. First, it is similar to bagging in using voting for classification or averaging for numeric prediction. Like bagging, boosting combines the models of the same type. However, boosting is iterative. "Whereas in bagging individual models are built separately, in boosting each new model is influenced by the performance of those built previously. Boosting encourages new models to become experts for instances handled incorrectly by earlier ones" [**?**]. Random forest, bagging and boosting develop an ensemble of base models and use them in combination. In our experiments, they demonstrate consistent performance in software quality prediction [19, 18].

Another way to improve the performance of fault prediction models is to use feature subset selection (FSS). FSS selects useful attributes or features and eliminate irrelevant or noisy features before learning process starts. Principal Component Analysis (PCA) is a classical method to cope with multicollinearity among attributes. PCA transforms a set of attributes (metrics, features) into their uncorrelated linear combinations. PCA selected nine major components from *38* software metrics on open source Apache 1.3 and 2.0 projects [8].

5. CONCLUSION

Software quality models offer tangible advantages for optimizing project's V&V activities by uncovering modules in which software faults are most likely to hide. From the methodological perspective, these are binary classification models. Typical proportion of modules which are likely to contain faults is rather small. This makes automated binary classification problem of detecting faulty modules more difficult.

In this paper, we analyzed the possible advantages of cost sensitive software quality modeling. Cost sensitive modeling assigns different cost factors to overlooking a faulty module and falsely tagging a fault free module as fault prone. By minimizing the overall cost of misclassification, rather than the number of misclassified modules, we expect to develop better classifiers.

We analyzed the impact of eleven different misclassification costs to software quality modeling, using the projects from the NASA MDP repository. Cost-sensitive modeling does not improve the overall performance of classification models. Nevertheless, explicit information about misclassification cost makes it easier for software managers to select the most appropriate model for their specific project environment. The alternative to cost-sensitive modeling is to determine the most appropriate threshold in a set of models developed in absence of cost information, which we believe to be more challenging. Our experiments further indicate that in projects where the exact misclassification cost is unknown, a likely scenario in practice, cost sensitive models with similar misclassification cost ratios are likely to exhibit performance which is not significantly different.

6. REFERENCES

[1] Metric data program. NASA Independent Verification and Validation facility, Available from `http://MDP.ivv.nasa.gov`.

[2] E. Arisholm, L. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE 2007),Sweden*, pages 215–224, 2007.

[3] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators, 1996.

[4] G. Boetticher. An assessment of metric contribution in the construction of a neural network-based effort estimator. In *Second International Workshop on Soft Computing Applied to Software Engineering, Enschade, NL*, 2001. Available from: `http://nas.cl.uh.edu/boetticher/publications.html`.

[5] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[6] V. U. Challagulla, F. B. Bastani, and I.-L. Yen. A unified framework for defect data analysis using the mbr technique. In *Proc. of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, pages 39–46, 2006.

[7] J. Demsar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7, 2006.

[8] G. Denaro and M. Pezze. An empirical evaluation of fault-proneness models. In *Proc. of International Conference on Software Engineering*, pages 241–251, 2002.

[9] P. Domingos. How to get a free lunch: A simple cost model for machine learning applications. In *Proc. AAAI98/ICML98, Workshop on the Methodology of Applying Machine Learning*, pages 1–7. AAAI Press, 1998.

[10] C. Drummond and R. C. Holte. Cost curves: An improved method for visualizing classifier performance. *Machine Learning*, 65(1):95–130, 2006.

[11] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. *icse*, 00:0329, 2001.

[12] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. PWS Publishing Company,International Thompson Press, 1997.

[13] R. J. Freund and W. J. Wilson. Regression analysis: Statistical modeling of a response variable. *Academic Press*, 1998.

[14] L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *Proc. of the 15th International Symposium on Software Relaibility Engineering ISSRE'04*, pages 417–428, 2004.

[15] M. H. Halstead. *Elements of Software Science*. Elsevier, North-Holland, 1975.

[16] T. Javed, M. e Maqsood, and Q. S. Durrani. A study to investigate the impact of requirements instability on software defects. *SIGSOFT Softw. Eng. Notes*, 29(3):1–7, 2004.

[17] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow. Comparing design and code metrics for software quality prediction. In *PROMISE '08: Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 11–18, New York, NY, USA, 2008. ACM.

[18] Y. Jiang, B. Cukic, and Y. Ma. Techniques for evaluating fault prediction models. *Empirical Softw. Engg.*, 13(5):561–595, 2008.

[19] Y. Jiang, B. Cukic, and T. Menzies. Fault prediction using early lifecycle data. pages 237–246. Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on, Nov. 2007.

[20] Y. Jiang, B. Cukic, and T. Menzies. Cost curve evaluation of fault prediction models. *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 197–206, Nov. 2008.

[21] K. Kaminsky and G. Boetticher. Building a genetical ly engineerable evolvable program (geep) using breath-based explicit knowledge for predicting software defects. 2004.

[22] T. Khoshgoftaar. An application of zero-inflated poisson regression for software fault prediction. In *Proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong*, pages 66–73, Nov 2001.

[23] T. M. Khoshgoftaar and E. B. Allen. Classification of fault-prone software modules: Prior probabilities,costs, and model evaluation. *Empirical Softw. Engg.*, 3(3):275–298, 1998.

[24] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Cost-benefit analysis of software quality models. *Software Quality Control*, 9(1):9–30, 2001.

[25] T. M. Khoshgoftaar and N. Seliya. Fault prediction modeling for software quality estimation: Comparing commonly used techniques. *Empirical Software Engineering*, 8(3):255–283, 2003.

[26] Y. Ma. *An Empirical Investigation of Tree Ensembles in Biometrics and Bioinformatics*. PhD thesis, January 2007.

[27] Y. Malaiya and J. Denton. Requirement volatility and defect density, 1999.

[28] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, January 2007. Available from `http://menzies.us/pdf/06learnPredict.pdf`.

[29] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *27th International Conference on Software Engineering, ICSE'05*, pages 284–292, May 2005.

[30] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 521–530, New York, NY, USA, 2008. ACM.

[31] N. Nagappanand, T. Ball, and B. Murphy. Using historical data and product metrics for early estimation of software failures. In *Proc. ISSRE , Raleigh, NC*, pages 62–71, 2006.

[32] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.

[33] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[34] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, 2003.

[35] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, Los Altos, US, 2005.