Focus Article

The use of decision trees for cost-sensitive classification: an empirical study in software quality prediction

Naeem Seliya[1] and Taghi M. Khoshgoftaar[2*]

This empirical study investigates two commonly used decision tree classification algorithms in the context of cost-sensitive learning. A review of the literature shows that the cost-based performance of a software quality prediction model is usually determined after the model-training process has been completed. In contrast, we incorporate cost-sensitive learning during the model-training process. The C4.5 and Random Forest decision tree algorithms are used to build defect predictors either with, or without, any cost-sensitive learning technique. The paper investigates six different cost-sensitive learning techniques: AdaCost, Adc2, Csb2, MetaCost, Weighting, and Random Undersampling (RUS). The data come from case study include 15 software measurement datasets obtained from several high-assurance systems. In addition, to a unique insight into the cost-based performance of defection prediction models, this study is one of the first to use misclassification cost as a parameter during the model-training process. The practical appeal of this research is that it provides a software quality practitioner with a clear process for how to consider (during model training) and analyze (during model evaluation) the cost-based performance of a defect prediction model. RUS is ranked as the best cost-sensitive technique among those considered in this study. *WIREs Data Mining Knowl Discov* 2011 1 448–459 DOI: 10.1002/widm.38

INTRODUCTION

Software quality prediction models are part of the tool box of techniques for improving the quality and reliability of software-based systems. The basic aim of such models is to assist the software quality assurance and/or testing team to pursue a targeted and cost-effective evaluation of program modules that are likely to be error-prone. The typical process in building a software quality prediction model begins by collecting specific software measurements and defect data for the different program modules of a previously developed similar system or earlier system release. The software metrics and defect data are then used to train a defect prediction model, which subsequently can predict the quality of program modules that are currently under development.

The quantitative software engineering nature of defect prediction requires the application of data mining and machine learning techniques, such as classification and quantitative prediction. In the context of software quality prediction during the practice of software engineering, classification models are commonly used to predict modules as either fault-prone (*fp*) or not-fault-prone (*nfp*). Some practitioners prefer to predict a quality-based ranking of program modules, e.g., ranking the program modules with respect to the predicted number of defects they are likely to have.[1] Various data mining and statistical techniques have been used for building defect prediction models.[2,3]

Software defect prediction models have been a frequent subject of research in recent years, with various models and approaches proposed in the literature.[4] Among the various learners used for defect prediction, decision tree algorithms are particularly attractive to practitioners. This is because of their white-box structure which allows the formation

*Correspondence to: taghi@cse.fau.edu

[1]Computer and Information Science, University of Michigan—Dearborn, Dearborn, MI, USA

[2]Computer and Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL, USA

DOI: 10.1002/widm.38

of rules that can be used to predict the quality of program modules. Such rules are well suited for knowledge extraction from software project repositories. This study focuses on using decision trees for defect prediction, especially in the context of cost-sensitive learning.

A binary classification problem (such as *fp* or *nfp*) has a confusion matrix consisting of four components, i.e., true positive, true negative, false positive, and false negative, in which the positive class represents *fp* modules and the negative class represents *nfp* modules. A false positive indicates an error in which an *nfp* program module is misclassified as *fp*, whereas a false negative indicates an error in which an *fp* program module is misclassified as *nfp*. In contrast, a true positive and a true negative respectively indicate the correct prediction of an *fp* and an *nfp* program module. From the point of view of software practitioners, it is clear that a false negative error is a more serious problem because it could lead to a lost opportunity to detect an actual *fp* module. On the other hand, a false positive error could imply spending valuable project resources to inspect and evaluate a program module that is already of high quality.

In the literature,[4] we find various strategies for evaluating the *goodness* or lack thereof of a given software quality prediction model. For example, misclassification error rates, overall accuracy, F-measure, Recall, Precision, etc. have been used to evaluate the performance of defect prediction models. However, as indicated previously, the consequences of misclassifying an *fp* program module is more severe than the cost of misclassifying an *nfp* program module. Consequently, a software practitioner is interested in the problem of incorporating the cost-based performance of software quality prediction models into the processes of model building and model evaluation. Assuming the availability of the respective costs of misclassifying *fp* and *nfp* modules, the total cost of misclassification (TC) [or expected cost of misclassification (ECM)] can, and has been, used as cost-based performance metric for defect predictors.[5] However, to the best of our knowledge, this has only been done post model training, i.e., total cost or expected cost is only computed during the model-evaluation phase. We emphasize in this paper that cost of misclassification should be used as one of the model parameters during the model-training process.

The primary objective of this study is to investigate several cost-sensitive learning techniques during the actual process of training a software quality prediction model. As far as we know, this study is one of the first to associate cost-sensitive learning with a given classification algorithm during the training process of building a defect predictor. Although we appreciate the usefulness of other performance metrics, such as error rate, receiver operating characteristic (ROC) curve, F-measure, Precision, Recall, etc., a software practitioner is very much interested in the cost associated with a given software quality estimation model. This study provides a clear solution to the practitioner with regards to training a defect predictor at a specific misclassification cost value.

The software quality prediction models were developed by using two commonly used decision tree algorithms, C4.5[6] and Random Forest,[7] either with, or without, one of the six cost-sensitive learning techniques. Software metrics and defect data from several software projects are used as case study data (15 software measurement datasets, in total) to conduct a large comparative study on six different cost-sensitive learning techniques, including AdaCost,[8] Csb2 ,[9] Adc2,[10] MetaCost,[11] Random Undersampling (RUS),[12] and Weighting.[6] The latter two are considered unconventional cost-sensitive learning techniques, thus adding to the uniqueness of this study. Several of the cost-sensitive techniques considered can be applied to any error-based classifier without the need to modify the classification algorithm itself. Although some of the cost-sensitive techniques were available in the data mining tool Weka,[6] the other techniques were implemented especially for this research. A cost-based-performance ranking of the cost-sensitive techniques from best to worst was observed as follows: RUS, MetaCost, Weighting, Csb2, Adc2, and AdaCost.

The remainder of this paper is structured as follows: section *Related Work* discusses some of the most relevant related works in the context of this work; section *Cost-Sensitive Learning Techniques* details the different cost-sensitive learning techniques used in our comparative evaluation; section *Software Measurement Data* provides a summary of the various software measurement datasets used as case studies; section *Performance Metrics* describes the different performance metrics used for evaluating the defect predictors; section *Empirical Case Study* provides details on the empirical work, including experimental settings and the presentation and analysis of the results; and section *Conclusion* concludes this study with a summary of our empirical investigation and results, and provides suggestions for future work.

RELATED WORK

In the literature, one can find several studies on software measurements-based quality estimation models;

hence, we refer the reader to the relevant cited references in this paper for a more in-depth coverage of software quality prediction models. However, in this section, we present an overview of literature that explore incorporating misclassification costs during software quality modeling and analysis. We note that research on incorporating misclassification costs during the training process of building a software quality prediction model is almost nonexistent. This study is unique in incorporating cost-sensitive learning during the training process of building a software quality model.

The ECM has been commonly used to evaluate the cost-based performance of a given software quality prediction (i.e., classifying program modules as *fp* or *nfp*) model.[3,13,14] In some cases that utilize ECM for a cost-based performance evaluation, the TC is normalized with respect to the number of instances (i.e., program modules) in the dataset. Much of the prior research in software quality modeling and analysis has focused on using ECM as a metric for evaluating software quality prediction models.[3,14] Although ECM offers an intuitive approach to evaluating the costs associated with using a defect prediction model, it is not incorporated or computed during the model-training process. It is important to note that this strategy is followed by most existing studies.

In our prior studies, we have investigated evolutionary techniques to train models that are optimized for multiple objectives, including model complexity and ECM.[15] The black-box, nontraditional, and computational complexity aspects of evolutionary techniques tend to limit their appeal to software quality practitioners. In addition, with ECM as the performance evaluation metric, one would have to evolve software quality models for different misclassification costs. Furthermore, the problem of relatively slow training times and the tedious task of optimizing evolutionary parameters makes genetic programming and genetic algorithms less attractive to analysts.

Drummond and Holte[16] recently introduced cost curves as a visual representation of the performance of two-group classifiers across all class distributions and misclassification costs. They state that ROC curves and cost curves are highly correlated, i.e., there is a bidirectional point/line duality between them. This implies that a point in ROC space is represented by a line in cost space and a line in ROC space is represented by a point in cost space, and vice versa. An inherent problem with using cost curves is that they are computed only during the model-evaluation (i.e., testing) phase and not during the model-training process of building the classifier.

Researchers, such as Jiang et al.,[13] have attempted to apply cost curves to the problem of building defect prediction models. Based on a study of multiple software measurement datasets, it was recommended that cost curves should be adopted as one of the standard methods for evaluating fault prediction models. However, as stated previously, cost curves are not incorporated into the actual training process of building a classifier. More specifically, cost curves are used as a performance metric during model evaluation.

In contrast to using cost curves for performance evaluation, we investigate using cost-sensitive learning techniques during the model-training process of building a software quality model. It is intuitive to realize that the software quality analyst is more interested in what specific factors are considered during the model-training process in addition to the model-evaluation process. As cost curves are highly correlated to ROC curves,[16] they tend to suffer from the same limitations that ROC curves suffer, i.e., poor practical appeal to the software quality analyst largely due to insufficient input on the application of the model-evaluation knowledge toward making vital quality improvement decisions.

A previous work by our team[17] investigated the cost-sensitive learning technique MetaCost for improving software defect prediction models. The present work expands and elaborates on this by exploring a wider range of cost-sensitive learning techniques and a more diverse collection of software metric datasets, as well as considering the Random Forest learner.

COST-SENSITIVE LEARNING TECHNIQUES

The theoretical foundations for cost-sensitive learning are introduced by Elkan.[12] We consider six different cost-sensitive learning techniques in the context of software quality modeling. These include three boosting-based techniques (AdaCost, Adc2, and Csb2), MetaCost, Weighting, and RUS. Some of these techniques were part of the Weka[6] data mining tool, whereas others were implemented specifically for our study. Most of these techniques can be applied to any error-based classifier without the need to modify the classification algorithm itself. We present a brief overview of each cost-sensitive technique in the remainder of this section.

Cost-Sensitive Boosting

Boosting is a metalearning technique that improves the performance of classifiers by iteratively building

an ensemble of classifiers with the goal of correctly classifying those examples (instances in the dataset) that were incorrectly classified during the previous iteration. AdaBoost[18] is the most popular boosting algorithm, and has been shown to improve the performance of any weak learner. However, AdaBoost is not designed for cost-sensitive learning, so various modifications to AdaBoost have been proposed for that purpose.

Sun et al.[10] present an in-depth examination of cost-sensitive boosting. The authors compare the performance of several cost-sensitive boosting techniques, and demonstrate that Adc2 (a variation of AdaBoost) results in better performance using most of their experimental datasets. In this study, we evaluate (among other cost-sensitive techniques) AdaCost,[8] Csb2,[9] and Adc2[10] in the context of software measurement datasets. Algorithmic details of AdaCost, Csb2, and Adc2 are omitted here for simplicity but can be found in the respective references.

AdaCost[8] (abbreviated as Adac) provides a cost-sensitive alternative to AdaBoost [9]. Although AdaBoost adjusts example costs without concern for the class of the example, AdaCost adjusts the example weights differently for different classes by introducing a 'misclassification cost adjustment function'. The degree of adjustment is controlled by a user supplied cost ratio.

The Csb2[9] approach is a cost-sensitive variation of AdaBoost,[18] and uses modified formulas for reweighting examples in order to minimize the number of expensive errors, and therefore the total cost.

The Adc2[10] technique also modifies the reweighting formula of AdaBoost,[18] and was presented as a cost-sensitive boosting algorithm for datasets suffering from class imbalance. Each of these cost-sensitive boosting algorithms takes the cost ratio into account when modifying an example's weight.

MetaCost

Proposed by Domingos[11] for making any error-based classifier cost sensitive, MetaCost is based on *Bayes optimal prediction*. If for a given example (instance) x, we know the probability of each class j, i.e., $P(j|x)$, the Bayes optimal prediction for x is the class i that minimizes the *conditional risk*, $R(i|x)$ [see Eq. (1)], which is the expected cost of predicting that x belongs to class i. The Bayes optimal prediction is certain to achieve the lowest possible overall cost over all possible examples x, weighted by their probabilities $P(x)$. $C(i, j)$ and $P(j|x)$ together with Eq. (1) imply a partition of the instance space X into j regions, such that class j is the optimal (i.e., lowest cost) prediction in region j[11]:

$$R(i|x) = \sum_j P(j|x)C(i, j). \quad (1)$$

MetaCost modifies the labels of training data instances so that their labels represents their 'optimal classes'. This is achieved by learning multiple classifiers, and using the result of each classifier as a vote in determining the probability that an instance belongs to a specific class. Bagging[19] is used to build an ensemble of learners. Samples (program modules) are taken, with replacement, from the training dataset. This is repeated m times (we use $m = 10$) with m models being trained using the resampled datasets. The probability that an instance (program module) belongs to a class is based on the fraction of votes it received, or an unweighted average of the probability estimates of the m models. Based on the estimated probability and the cost ratio, new class labels are assigned. The newly labeled training dataset is then used by the given classification algorithm to produce a cost-sensitive software quality prediction model. We differ further algorithmic details to the work of Domingos.[11]

Weighting

Weka[6] provides a metaclassifier called 'CostSensitiveClassifier' which can be used to adjust the weights of examples of each class based on a supplied cost matrix. This cost-sensitive technique is referred to as Whtg throughout this paper. Weights (W_i) are assigned to each example (instance) of class i using the formula,

$$W_i = B_i \frac{C}{D}, \quad (2)$$

in which C is the number of examples in the training dataset (if another metaclassifier, such as boosting, is used this is actually the sum of the weights of all examples in the training data), $B_i = C(j, i)$, and D is the sum of all costs of all examples in the training data based on the given cost matrix.

Example weighting does result in different models for different cost ratios; hence, new models must be built whenever the cost matrix is changed. Every example gets the exact same increased (or decreased) weight using Weighting, and no data points are lost during this process. However, the value of every example of the less expensive class is diminished, and the value of every example of the more expensive class is increased identically. This technique does require that the base classifier be implemented in such a way that example weights can be taken into account.

Random Undersampling

Elkan[12] provides a method for making optimal decisions by rebalancing the training data. The strategy behind data sampling is to decrease (undersampling) or increase (oversampling) the number of instances in the training data based on the cost ratio, i.e., $CR = C(0, 1)/C(1, 0)$, where $C(0, 1)$ is the cost of misclassifying an *fp* module as *nfp*, and $C(1, 0)$ is the cost of misclassifying an *nfp* module as *fp*. Considering the default decision threshold of 0.5, undersampling decreases the number of majority class instances (i.e., *fp* modules) by a factor of $1/CR$, whereas oversampling increases the number of minority class instances (i.e., *nfp* modules) by a factor of CR.

In our study, we consider RUS as the undersampling cost-sensitive technique. RUS modifies the training data distribution by randomly removing instances of the majority class. With regards to cost-sensitive learning, this reduces the importance the learner will place on majority class instances; hence, increasing the importance placed on the minority class instances. Despite the running the risk of removing important information from the training data, RUS has been shown to perform very well when training data is imbalanced.[20] Thus, RUS achieves cost-sensitive learning by modifying two key parameters, i.e., the cost ratio and the size of the majority class.

SOFTWARE MEASUREMENT DATA

Our empirical case study was conducted with 15 software measurement datasets, consisting of software metrics and defect data obtained from various real-world software systems. The basic statistics for the different software measurement datasets are shown in Table 1. For a given dataset, the table lists the number of software metrics, number of fault-prone program modules, number of not-fault-prone modules, total number of modules, and the proportion of *fp* modules in the dataset. We selected a wide range of dataset sizes, with 282 modules being the smallest (CCCS) and 8850 modules being the largest (JM1). In addition, the datasets show a wide range for the relative proportion of *fp* modules, with the smallest being 1.30% (SP3) and the largest being 29.43% (CCCS-2). The use of the specific software metrics in our study does not advocate their effectiveness—a different project may consider a different set of software measurements for analysis.[4,21]

The following list summarizes the different high-assurance software systems and their software measurement datasets used in our case studies:

TABLE 1 | Software Measurement Datasets

	# metrics	# *fp*	# *nfp*	# total	% *fp*
SP1	42	230	3419	3649	6.30
SP2	42	189	3972	3981	4.75
SP3	42	46	3495	3541	1.30
SP4	42	92	3886	3978	2.31
CCCS-2	8	83	199	282	29.43
CCCS-4	8	55	227	282	19.50
CCCS-8	8	27	255	282	9.57
CCCS-12	8	16	266	282	5.67
CM1	13	48	457	505	9.50
JM1	13	1687	7163	8850	19.06
KC1	13	325	1782	2107	15.42
KC2	13	106	414	520	20.38
KC3	13	43	415	458	9.39
MW1	13	31	372	403	7.69
PC1	13	76	1031	1107	6.87

1. The SP1, SP2, SP3, and SP4 datasets represent four successive releases of a large legacy telecommunications system, and the case study data was based on 42 software metrics which included 24 product metrics, 14 process metrics, and four execution metrics.[3] The software system is an embedded-computer application that included finite-state machines. Using the procedural development paradigm, the software was written in PROTEL (a high-level programming language) and was maintained by professional programmers in a large organization. Fault data was collected at the module-level by the problem reporting system. A module was considered as *nfp* if it had no postrelease faults, and *fp* otherwise. The number of program modules in the four datasets are: 3649 for SP1, 3981 for SP2, 3541 for SP2, and 3978 for SP4. The relative distribution of the *fp* and *nfp* modules for SP1, SP2, SP3, and SP4 are shown in Table 1.
2. The JM1 project, written in C, is a large project for a real-time ground system that uses simulations to generate predictions for missions. The JM1 dataset consisted of 10,883 program modules, of which 2105 modules had software defects (ranging from 1 to 26) whereas the remaining 8778 modules were defect-free. The dataset contained some inconsistent modules, i.e., those with identical software measurements but with

different class labels.[22] Upon removing such modules, the dataset was reduced to 8850 modules, consisting of 1687 *fp* modules (i.e., with one or more defects) and 7163 *nfp* modules (i.e., with no defects). This number-of-defects-based definition of *fp* and *nfp* program modules also applies to the CM1, KC1, KC2, KC3, MW1, and PC1 datasets.

Each program module in the JM1 datasets was characterized by 13 basic software product metrics.[23] These same metrics were also used for the CM1, KC1, KC2, KC3, MW1, and PC1 datasets, and they were primarily governed by their availability, internal workings of the projects, and the data collection tools used. The type and numbers of metrics made available were solely determined by the NASA Metrics Data Program. Other metrics, including software process and object-oriented metrics, were not available at the time of modeling and analysis.

3. The KC1 project is a single CSCI (Computer Software Configuration Item) within a large ground system and consists of 43 KLOC (thousand lines of code) of C++ code. A given CSCI comprises of logical groups of computer software components (CSCs). The dataset contains 2107 modules, of which 325 are *fp* and 1782 are *nfp*. The maximum number of faults in a program module is 7.
4. The KC2 project, written in C++, is the science data processing unit of a storage management system used for receiving and processing ground data for missions. The dataset includes only those modules that were developed by NASA software developers and not COTS (Commercial-Of-The-Shelf) software. The dataset contains 520 modules, of which 106 are *fp* and 414 are *nfp*. The maximum number of faults in a software module is 13.
5. The KC3 project, written in 18 KLOC of Java, is a software application that collects, processes, and delivers satellite metadata. The dataset contains 458 modules, of which 43 are *fp* and 415 are *nfp*. The maximum number of faults in a module is 6.
6. The CM1 software measurement dataset is that of a science instrument application written in C Code with approximately 20 KLOC. The dataset contains 505 modules, of which 48 are *fp* and 457 are *nfp*. The maximum number of faults in a module is 5.
7. The MW1 project is a software application from a zero gravity combustion experiment that has been completed. The MW1 dataset consists of 8000 lines of C code. The dataset contains 403 modules, of which 31 are *fp* and 372 are *nfp*. The maximum number of faults in a module is 4.
8. The PC1 project is a flight software from an earth orbiting satellite that is no longer operational. It consists of 40 KLOC of C code. The software measurement dataset contains 1107 modules, of which 76 are *fp* and 1031 are *nfp*. The maximum number of faults in a module is 9.
9. The CCCS-2, CCCS-4, CCCS-8, and CCCS-12 datasets represent software metrics and defect data for a large military command, control, and communications system (CCCS).[24] The CCCS datasets are based on software product metrics. The numerical suffix for the CCCS project represents the number of defects threshold used for determining whether a program module is considered *fp* or *nfp*.

PERFORMANCE METRICS

In this study, the performance of the two classifiers (software quality prediction models) is evaluated primarily by comparing their *per-example-cost* (PEC), which is the TC divided by the number of instances in the dataset. TC is given by

$$TC = \#fpos \times C(1,0) + \#fneg \times C(0,1), \quad (3)$$

in which $\#fpos$ is the number of *nfp* modules predicted as *fp* and $\#fneg$ is the number of *fp* modules predicted as *nfp*. As the exact costs of misclassifications are unknown during modeling and analysis, different values for the cost ratio, i.e., $C(0,1)/C(1,0)$, are used to provide the analyst with a broad range of the performance space.

We also present some commonly used metrics for classifier evaluation. For a two-group classification problem such as *fp* (i.e., positive) or *nfp* (i.e., negative), there can be four possible outcomes of classifier prediction: true positive (*TP*), false positive (*FP*), true negative (*TN*), and false negative (*FN*). A *TP* outcome involves the correct prediction of an *fp* module, whereas a *TN* outcome involves the correct prediction of an *nfp* module. An *FP* prediction occurs when an *nfp* module is incorrectly predicted as *fp*, whereas

TABLE 2 | Average Performance of Techniques at CR = 10

	Adac2	Adac	Csb2	Meta	RUS	Whtg	None
TPR	0.6027	0.4751	0.4609	0.5595	0.7369	0.4860	0.3121
TNR	0.6870	0.8914	0.8616	0.8689	0.7235	0.8849	0.9616
FPR	0.3130	0.1086	0.1384	0.1311	0.2765	0.1151	0.0384
FNR	0.3973	0.5249	0.5391	0.4405	0.2631	0.5140	0.6879
ACR	0.7036	0.8707	0.8410	0.8562	0.7442	0.8652	0.9047
FM	0.3237	0.3958	0.3635	0.4239	0.3705	0.3934	0.3364
GM	47.43	55.98	52.27	63.78	69.48	57.22	43.78
TC	134.26	110.21	128.11	102.97	92.38	112.29	158.47
PEC	0.0568	0.0486	0.0553	0.0445	0.0419	0.0488	0.0669

an *FN* prediction occurs when an *fp* module is incorrectly predicted as *nfp*.

A two-by-two confusion matrix depicts the numbers of instances predicted by each of the four possible outcomes: number of true positives (*#TP*), number of true negatives (*#TN*), number of false positives (*#FP*), and number of false negatives (*#FN*). Based on these four values, different performance metrics can be computed. If, for a given dataset, N is the total number of program modules, N_{fp} is the number of *fp* instances, and N_{nfp} is the number of *nfp* instances, then

$$TPR = \frac{\#TP}{N_{\text{fp}}}, \tag{4}$$

$$TNR = \frac{\#TN}{N_{\text{nfp}}}, \tag{5}$$

$$FPR = \frac{\#FP}{N_{\text{nfp}}}, \tag{6}$$

$$FNR = \frac{\#FN}{N_{\text{nfp}}}, \tag{7}$$

$$ACR = \frac{\#TP + \#TN}{N}, \tag{8}$$

$$FM = \frac{2 \times \#TP}{\#TP + \#FP + \#FN}, \tag{9}$$

$$GM = \sqrt{TPR \times TNR}, \tag{10}$$

where *TPR* is the true positive rate, *TNR* is the true negative rate, *FPR* is the false positive rate, *FNR* is the false negative rate, *ACR* is the overall accuracy rate, *FM* is the F-measure, and *GM* is the geometric mean. The latter two metrics are commonly used performance metrics in the data mining and machine learning community.[6]

EMPIRICAL CASE STUDY

Experimental Settings

The defect prediction models are built using two commonly used decision tree algorithms: C4.5 and Random Forest. In the case of C4.5, we used the default parameter settings in Weka.[6] In the case of random forest, we change the number of subtrees to 100 from 10 (default) trees, as that improves the classifier performance. We refer to this variant as RF100.

The cost ratios used include 10, 15, 20, 25, 30, 40, and 50. Generally speaking, for high-assurance systems such as those of our case study, a cost ratio of 25 is considered appropriate. In order to provide the analyst with a better insight into cost-sensitive software quality modeling, we consider a wide range of cost ratio values. One may consider a different set of cost ratio values depending on their appropriateness to the software project under consideration.

A classifier is built using 10 fold cross-validation, and this process is repeated 10 times (i.e., 10 runs). The empirical results (shown in the next section) represent the classifier performance averages across the 10 runs. Thus, with 15 datasets, two learners, seven cost-sensitive learning techniques (six techniques and None), seven cost ratios, 10 cross-validation folds, and 10 runs, a total of 147,000 combinations of models/cost ratio were constructed and evaluated. This signifies the magnitude of the scale of our empirical study on cost-sensitive defect prediction.

Results And Analysis

The average performances of the two decision tree algorithms over all case study datasets are presented in Tables 2–8, where each table corresponds to a given cost ratio. Although our primary performance metrics

TABLE 3 Average Performance of Techniques at CR = 15

	Adac2	Adac	Csb2	Meta	RUS	Whtg	None
TPR	0.6142	0.5089	0.4953	0.6202	0.7956	0.5159	0.3121
TNR	0.6476	0.8674	0.8298	0.8292	0.6546	0.8618	0.9616
FPR	0.3524	0.1326	0.1702	0.1708	0.3454	0.1382	0.0384
FNR	0.3858	0.4911	0.5047	0.3798	0.2044	0.4841	0.6879
ACR	0.6679	0.8548	0.8180	0.8289	0.6887	0.8496	0.9047
FM	0.3074	0.3964	0.3604	0.4199	0.3459	0.3919	0.3364
GM	43.48	57.60	52.60	66.29	69.00	58.38	43.78
TC	197.17	142.43	174.12	121.94	107.21	143.97	235.40
PEC	0.0757	0.0644	0.0738	0.0547	0.0501	0.0646	0.0988

TABLE 4 Average Performance of Techniques at CR = 20

	Adac2	Adac	Csb2	Meta	RUS	Whtg	None
TPR	0.6190	0.5331	0.5197	0.6695	0.8346	0.5422	0.3121
TNR	0.6232	0.8454	0.8044	0.7845	0.5990	0.8410	0.9616
FPR	0.3768	0.1546	0.1956	0.2155	0.4010	0.1590	0.0384
FNR	0.3810	0.4669	0.4803	0.3305	0.1654	0.4578	0.6879
ACR	0.6456	0.8384	0.7995	0.7956	0.6433	0.8353	0.9047
FM	0.2984	0.3929	0.3575	0.4106	0.3275	0.3894	0.3364
GM	40.48	58.60	52.23	66.68	67.30	59.19	43.78
TC	254.88	170.18	215.62	132.44	117.78	168.64	312.32
PEC	0.0937	0.0794	0.0899	0.0624	0.0555	0.0775	0.1307

TABLE 5 Average Performance of Techniques at CR = 25

	Adac2	Adac	Csb2	Meta	RUS	Whtg	None
TPR	0.6172	0.5398	0.5293	0.7074	0.8544	0.5667	0.3121
TNR	0.6117	0.8285	0.7874	0.7369	0.5581	0.8169	0.9616
FPR	0.3883	0.1715	0.2126	0.2631	0.4419	0.1831	0.0384
FNR	0.3828	0.4602	0.4707	0.2926	0.1456	0.4333	0.6879
ACR	0.6337	0.8234	0.7862	0.7580	0.6089	0.8180	0.9047
FM	0.2908	0.3820	0.3503	0.3953	0.3137	0.3867	0.3364
GM	38.86	58.33	51.59	65.49	65.28	59.76	43.78
TC	316.37	234.50	255.50	142.43	125.75	187.83	389.24
PEC	0.1122	0.0992	0.1065	0.0688	0.0608	0.0888	0.1626

of interest are the total cost (TC) and per-example-cost (PEC), we present other performance metrics for completeness sake. These include true positive rate (TPR), true negative rate (TNR), false positive rate (FPR), false negative rate (FNR), overall accuracy (ACR), F-measure (FM), and geometric mean (GM). The tables present the results for each of the six cost-sensitive learning techniques, and for the models built without (denoted as None) any cost-sensitive learning techniques.

The cost of misclassification increases as the cost ratio increases, as expected. Moreover, the TPR (correct detection of *fp* program modules) increases with the cost ratio, as it should when a cost-sensitive technique is involved. With respect to the Geometric Mean performance metric, the defect prediction

TABLE 6 Average Performance of Techniques at CR = 30

	Adac2	Adac	Csb2	Meta	RUS	Whtg	None
TPR	0.6133	0.5393	0.5531	0.7417	0.8728	0.5870	0.3121
TNR	0.6047	0.8227	0.7684	0.6906	0.5200	0.7962	0.9616
FPR	0.3953	0.1773	0.2316	0.3094	0.4800	0.2038	0.0384
FNR	0.3867	0.4607	0.4469	0.2583	0.1272	0.4130	0.6879
ACR	0.6267	0.8162	0.7726	0.7203	0.5772	0.8030	0.9047
FM	0.2882	0.3718	0.3502	0.3817	0.3007	0.3828	0.3364
GM	37.30	58.05	51.80	63.49	63.31	59.96	43.78
TC	375.68	333.01	289.92	151.37	132.17	204.64	466.17
PEC	0.1305	0.1223	0.1183	0.0734	0.0644	0.0984	0.1945

TABLE 7 Average Performance of Techniques at CR = 40

	Adac2	Adac	Csb2	Meta	RUS	Whtg	None
TPR	0.5911	0.5356	0.5787	0.7866	0.8950	0.6219	0.3121
TNR	0.6208	0.8170	0.7368	0.6221	0.4708	0.7623	0.9616
FPR	0.3792	0.1830	0.2632	0.3779	0.5292	0.2377	0.0384
FNR	0.4089	0.4644	0.4213	0.2134	0.1050	0.3781	0.6879
ACR	0.6389	0.8074	0.7487	0.6641	0.5350	0.7781	0.9047
FM	0.2869	0.3539	0.3454	0.3590	0.2863	0.3756	0.3364
GM	36.09	57.17	50.69	59.38	60.20	60.03	43.78
TC	494.23	468.60	356.84	164.82	142.05	229.09	620.01
PEC	0.1707	0.1707	0.1423	0.0809	0.0701	0.1130	0.2583

TABLE 8 Average Performance of Techniques at CR = 50

	Adac2	Adac	Csb2	Meta	RUS	Whtg	None
TPR	0.5803	0.5533	0.5967	0.8178	0.9142	0.6479	0.3121
TNR	0.6265	0.8021	0.7177	0.5743	0.4263	0.7397	0.9616
FPR	0.3735	0.1979	0.2823	0.4257	0.5737	0.2603	0.0384
FNR	0.4197	0.4467	0.4033	0.1822	0.0858	0.3521	0.6879
ACR	0.6426	0.7951	0.7334	0.6245	0.4967	0.7613	0.9047
FM	0.2850	0.3457	0.3419	0.3425	0.2757	0.3724	0.3364
GM	35.50	57.38	50.25	56.82	56.71	60.17	43.78
TC	620.92	567.56	423.84	172.75	148.56	254.65	773.86
PEC	0.2106	0.2059	0.1674	0.0866	0.0738	0.1257	0.3221

models built in association with either MetaCost or RUS perform generally better than the other models. However, with respect to the F-measure performance metric, only models built with MetaCost perform better than the other models. The six cost-sensitive techniques are effective at reducing the total cost of misclassification compared to modeling without cost-sensitive learning. Among the different cost-sensitive techniques, RUS followed by MetaCost provide the most reduction in total cost of misclassification.

To further validate our observation on the better cost-based performances of RUS and MetaCost, we performed a one-way ANOVA (Analysis of variance) statistical test.[25] The goal of the test is to observe if the cost-based performances of the different cost-sensitive techniques are significantly different from

TABLE 9 | ANOVA Test Results

Source	Sum Sq.	d.f.	Mean Sq.	F	*P*-value
A	193.15	6	32.19	2870.24	0
Error	1648.65	146993	0.01		
Total	1841.81	146999			

each other. Hence, the PEC is used as the response variable for the ANOVA test. The test is performed across all datasets, across all cost ratios, and across both classification algorithms. The ANOVA test results are summarized in Table 9, where the *P*-value shows that the different cost-sensitive techniques, including None, are significantly different than each other. The assumptions of the ANOVA model were validated.

Subsequent to the positive findings of the ANOVA test, we conducted a multiple comparison using Tukey's Honestly Significant Difference (HSD) test.[25] The multiple comparisons were performed at a significance level of $\alpha = 0.05$. The ranking obtained by Tukey's HSD test is shown in Figure 1. The figure clearly shows that RUS outperforms the other six techniques (which includes None). A complete ranking of the seven techniques from best to worst is as follows: RUS, Meta, Whtg, Csb2, Adc2, Adac, and None.

In the interest of noting which of two decision tree algorithms generally performed better and other trends, we performed a four-way ANOVA test. The four factors are Factor A, the 15 case study datasets; Factor B, the two decision tree classification algorithms; Factor C, the seven cost ratios; and Factor D, the seven cost-sensitive techniques (including None). Once again the PEC values of the models is used as the response variable for the ANOVA test. It was found that the two decision tree algorithms were significantly different at a significance level of $\alpha = 0.05$. A subsequent multiple comparison based on Tukey's HSD test at $\alpha = 0.05$ found that the C4.5 learner yielded better (i.e., lower) PEC values than the RF100 learner. We note that the four-way ANOVA results and subsequent multiple comparison results are not shown due to paper-size considerations.

CONCLUSION

Commonly used decision tree algorithms, C4.5 and Random Forest, are used in a large comparative study in which the cost-based evaluation of software quality models is performed during the model-training process, instead of the commonly used approach of analyzing their cost-based performance during the model-evaluation phase. Six different cost-sensitive learning techniques were examined, and defect predictors were built with, and without, cost-sensitive learning. The case study data was obtained from 15 software

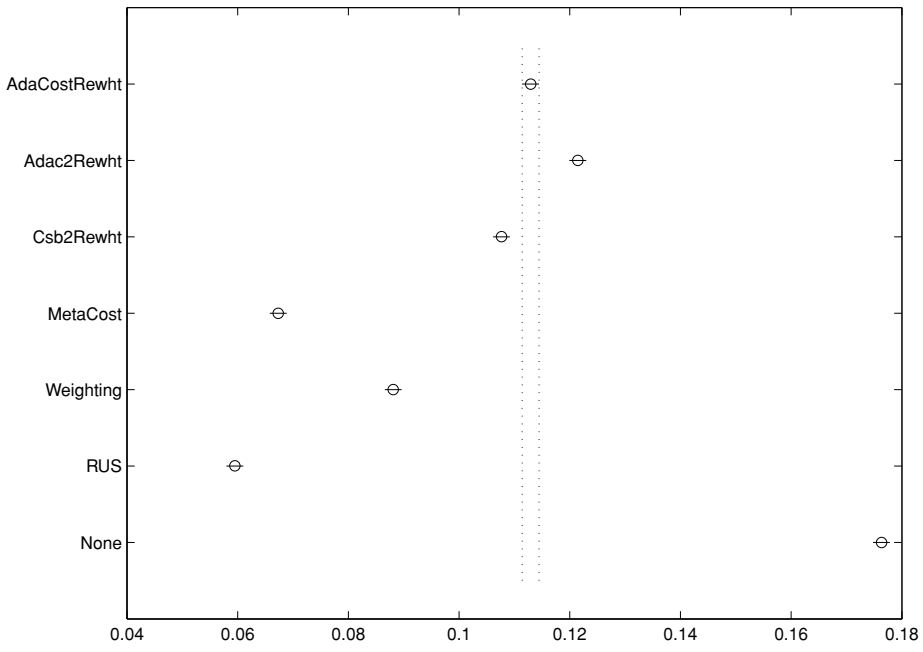

FIGURE 1 | Tukey's Honestly Significant Difference multiple comparisons.

measurement datasets that were collected from several software projects. To our knowledge, such an investigative study on the cost-based performance of software quality prediction models is quite unique.

The key results/conclusions are summarized as follows:

- The models built by including a cost-sensitive technique during the training process provided lower total costs of misclassification as compared to corresponding models built without any cost-sensitive learning—an intuitively expected empirical result.
- Among the six different cost-sensitive learning techniques investigated, RUS significantly outperformed the other techniques. A cost-based performance ranking of the cost-sensitive techniques from best to worst is as follows: RUS, Meta, Whtg, Csb2, Adc2, Adac, and None.
- In association with a cost-sensitive learning technique, the C4.5 decision tree algorithm is shown to perform significantly better than the Random Forest decision tree algorithm. However, it is known from the literature that the performance of a classification algorithm is affected by the various characteristics of the training data as well as the application domain.
- Based on the experience of this empirical study, the authors conclude that a software quality practitioner should strongly consider the cost ratio as a modeling parameter during the training process of building a defect predictor. However, this study does not advocate, in the context of software defect predictors, ignoring existing performance metrics such as accuracy, F-measure, Recall, and Precision, for example.

Future research directions are as follows: considering software measurement data from non-high-assurance systems, investigating other classification algorithms, evaluating other cost-sensitive learning techniques within the framework on this study, and focusing on strategies that make it easier and simpler for the software quality analyst to select a good defect predictor within practical constraints of a given project, such as lack of knowledge on misclassification costs, insufficient heuristic data, and unfamiliarity with the project's domain.

REFERENCES

1. Khoshgoftaar TM, Cukic B, Seliya N. An empirical assessment on program module-order models. *Qual Technol Quant Manag* 2007, 4:171–190.
2. Emam KE, Benlarbi S, Goel N, Rai SN. Comparing case-based reasoning classifiers for predicting high-risk software componenets. *J Syst Softw* 2001, 55:301–320.
3. Khoshgoftaar TM, Seliya N. Comparative assessment of software quality classification techniques: an empirical case study. *Empir Softw Eng J* 2004, 9:229–257.
4. Lessmann S, Baesens B, Mues C, Pietsch S. Benchmarking classification models for software defect prediction: a proposed framework and novel findings. *IEEE Trans Softw Eng* 2008, 34:485–496.
5. Liu Y, Khoshgoftaar TM, Seliya N. Evolutionary optimization of software quality modeling with multiple repositories. *IEEE Trans Softw Eng* 2010, 36:852–864.
6. Witten IH, Frank E. *Data Mining: Practical Machine Learning Tools and Techniques*. 2nd ed. San Francisco, CA: Morgan Kaufmann; 2005.
7. Breiman L. Random forests. *Mach Learn* 2001, 45:5–32.
8. Fan W, Stolfo SJ, Zhang J, Chan PK. Adacost: misclassification cost-sensitive boosting. In: *Proceedings of 16th International Conference on Machine Learning*. San Francisco, CA: Morgan Kaufmann; 1999, 97–105.
9. Ting KM. A comparative study of cost-sensitive boosting algorithms. In: *Proceedings of 17th International Conference on Machine Learning*. Stanford, CA: Morgan Kaufmann; 2000, 983–990.
10. Sun Y, Kamel MS, Wong AKC, Wang Y. Cost-sensitive boosting for classification of imbalanced data. *Pattern Recognit* 2007, 40:3358–3378.
11. Domingos P. Metacost: a general method for making classifiers cost-sensitive. In: *Proceedings of Knowledge Discovery and Data Mining*. New York: ACM Press; 1999, 155–164.
12. Elkan C. The foundations of cost-sensitive learning. In: *Proceedings of the 17th International Joint Conference on Artificial Intelligence*. Vol. 2. San Francisco, CA: Morgan Kaufmann Publishers Inc.; 2001, 973–978.
13. Jiang Y, Cukic B, Menzies T. Cost curve evaluation of fault prediction models. In: *Proceedings of the 19th International Symposium on Software Reliability*

Engineering. Seattle, WA: IEEE Computer Society; 2008, 197–206.

14. Khoshgoftaar TM, Seliya N, Herzberg A. Resource-oriented software quality classification models. *J Syst Softw* 2005, 76:111–126.
15. Khoshgoftaar TM, Liu Y, Seliya N. A multi-objective module-order model for software quality enhancement. *IEEE Trans Evolution Comput* 2004, 8:593–608.
16. Drummond C, Holte RC. Cost curves: an improved method for visualizing classifier performance. *Mach Learn* 2006, 65:95–130.
17. Seliya N, Khoshgoftaar TM. Value-based software quality modeling. In: *SEKE*. Skokie, IL: Knowledge Systems Institute Graduate School; 2009, 116–121.
18. Freund Y, Schapire R. Experiments with a new boosting algorithm. In: *Proceedings of 13th International Conference on Machine Learning*. Bari: Morgan Kaufmann; 1996, 148–156.
19. Breiman L. Bagging predictors. *Mach Learn* 1996, 26:123–140.
20. Van Hulse J, Khoshgoftaar TM, Napolitano A. Experimental perspectives on learning from imbalanced data. In: *Proceedings of the 24th International Conference on Machine Learning*. New York: ACM Press; 2007, 935–945.
21. Sayyad Shirabad J, Menzies TJ. The PROMISE repository of software engineering databases, School of Information Technology and Engineering, University of Ottawa, Canada; 2005.
22. Khoshgoftaar TM, Zhong S, Joshi V. Noise elimination with ensemble-classifier filtering for software quality estimation. *Intell Data Anal: An Int J* 2005, 9:3–27.
23. Seliya N, Khoshgoftaar TM. Software quality analysis of unlabeled program modules with semi-supervised clustering. *IEEE Trans Syst Man Cybern* 2007, 37:201–211.
24. Khoshgoftaar TM, Allen EB. Logistic regression modeling of software quality. *Int J Reliab Qual Saf Eng* 1999, 6:303–317.
25. Berenson ML, Levine DM, Goldstein M. *Intermediate Statistical Methods and Applications: A Computer Package Approach*. Englewood Cliffs, NJ: Prentice-Hall, Inc.; 1989.

