Paper Title: Addressing class imbalance and cost sensitivity in software defect prediction by combining domain costs and balancing costs

Author/s: Siers, M., & Islam, Md. Z.

Conference Title: Advanced Data Mining and Applications (ADMA) 12th International Conference

Dates held: 12/12/2016 – 15/12/2016 **Place**: Gold Coast, Queensland, Australia

Pages: 156-171

Abstract: Effective methods for identification of software defects help minimize the business costs of software development. Classification methods can be used to perform software defect prediction. When cost-sensitive methods are used, the predictions are optimized for business cost. The data sets used as input for these methods typically suffer from the class imbalance problem. That is, there are many more defect-free code examples than defective code examples to learn from. This negatively impacts the classifier's ability to correctly predict defective code examples. Cost-sensitive classification can also be used to mitigate the affects of the class imbalance problem by setting the costs to reflect the level of imbalance in the training data set. Through an experimental process, we have developed a method for combining these two different types of costs. We demonstrate that by using our proposed approach, we can produce more cost effective predictions than several recent cost-sensitive methods used for software defect prediction. Furthermore, we examine the software defect prediction models built by our method and present the discovered insights.

DOI: https://doi.org/10.1007/978-3-319-49586-6_11

Addressing Class Imbalance and Cost Sensitivity in Software Defect Prediction by Combining Domain Costs and Balancing Costs

Michael J. Siers(✉) and Md Zahidul Islam

School of Computing and Mathematics, Charles Sturt University, Bathurst, Australia
{msiers,zislam}@csu.edu.au

Abstract. Effective methods for identification of software defects help minimize the business costs of software development. Classification methods can be used to perform software defect prediction. When cost-sensitive methods are used, the predictions are optimized for business cost. The data sets used as input for these methods typically suffer from the class imbalance problem. That is, there are many more defect-free code examples than defective code examples to learn from. This negatively impacts the classifier's ability to correctly predict defective code examples. Cost-sensitive classification can also be used to mitigate the affects of the class imbalance problem by setting the costs to reflect the level of imbalance in the training data set. Through an experimental process, we have developed a method for combining these two different types of costs. We demonstrate that by using our proposed approach, we can produce more cost effective predictions than several recent cost-sensitive methods used for software defect prediction. Furthermore, we examine the software defect prediction models built by our method and present the discovered insights.

Keywords: Cost sensitive · Software defect prediction · Class imbalance

1 Introduction

Software defect prediction (SDP) is the task of predicting which sections of code are likely to be defective. Within the literature, these sections of code are referred to as modules. For a module to be considered as defective, it must contain at least one bug. Each function/method is commonly considered as one module. SDP has been studied in many areas of software such as web applications [12], mobile applications [15] and SDP for a specific programmer [9].

SDP can be performed both cost-sensitively and non-cost-sensitively. When performed non-cost-sensitively, the SDP system aims to make as many correct predictions as possible. That is, it is optimizing for accuracy. However, incorrectly predicting a defective module as defect-free is very costly for a software development business. A cost-sensitive SDP system is aware of this cost and is less likely to predict a module as defect-free when it is actually defective.

J. Li et al. (Eds.): ADMA 2016, LNAI 10086, pp. 156–171, 2016.
DOI: 10.1007/978-3-319-49586-6_11

Table 1. SDP costs

Cost	Business scenario	Typical value
TP	A defective module is correctly predicted as defective. The software development business must now assign resources to fixing the defective module	$1000
TN	A defect-free module is correctly predicted as containing no defects. No action is required	$0
FP	A defect-free module is incorrectly predicted as containing defects. After assigning resources to fix the module, the software development business learns that the module is actually defect-free. The typical value associated with this cost is the same as TP	$1000
FN	A defective module is incorrectly predicted as defect-free. This means that the defect will remain within the program. This can cause lots of damage to the program and potentially hinder the software development process at a later stage. The typical value associated with this cost is 5–8 times as much the cost of TP	$5000 to $8000

Typically, four costs need to be defined for a cost-sensitive SDP system. These are true positive (TP), true negative (TN), false positive (FP) and false negative (FN). An explanation of each of these costs is given in Table 1. The table includes typical values for these costs which are based on previous studies in SDP [10,16,18,19].

To build a software defect prediction system, a classification method such as C4.5 [13], EXPLORE [7], or SysFor [8] is used. These classification methods require a training dataset D_T as input. D_T is comprised of a set of records $R = \{R_0, R_1, \ldots, R_{|R-1|}\}$ and a set of attributes $A = \{A_0, A_1, \ldots, A_{|A-1|}\}$. Each record is comprised of a set of values $R_i = \{V_0, V_1, \ldots, V_{|A-1|}\}$ where each value corresponds to an attribute in A. Therefore, D_T can be thought of a table where each $R_i \in R$ is a row and each $A_j \in A$ is a column. One of the $A_j \in A$ is considered as the class attribute. This class attribute is the value of interest and is the value that the trained classifier will predict for a new, unseen record. In the case of SDP each $R_i \in R$ represents a module, each $A_j \in A$ represents a software measure, and the class attribute is whether or not the module is defective.

A D_T used to train a SDP system often suffers from the class imbalance problem. When a D_T contains many more records with one class value than records with a different class value, it is considered to be class imbalanced. More specifically, a SDP D_T typically contains many more examples of defect-free modules than defective modules. In this case, we refer to the defect class as the minority class, and the defect-free class as the majority class. This class imbalance problem causes a SDP system to have poor prediction performance on defective modules. Thus, a traditional classification method is more likely to

make many FN predictions which is very costly to a business as described in Table 1.

One method for dealing with the class imbalance problem is to use a cost-sensitive classification method. This is done by setting the value of the FN cost to be I times the value of the FP cost [20]. I is the number of majority class records divided by the number of minority class records. Since there should be no penalty for a correct prediction, the values of TP and TN costs are set to 0. This way, the costs reflect the level of imbalance. A cost-sensitive classifier trained using these costs will produce non-cost-sensitive predictions, but will not be biased towards predicting records as the majority class. In this study, we refer to these cost values as balancing costs. On the other hand, we refer to the cost values described in Table 1 as the domain costs.

In Sect. 3, we first propose a cost-sensitive framework which uses both balancing costs and domain costs. We then experiment different possible methods for combining a balancing cost-matrix and a domain cost-matrix into a single cost-matrix for creating a cost-sensitive SDP system. Our framework consists of a recent cost-sensitive classification method CSForest [18,19], and a recent method for calculating cluster-specific balancing costs named Standoff [20]. The framework also uses a recent class-specific clustering method named RBClust [21] as input for Standoff. In Sect. 4 this framework is compared against recent cost-sensitive methods proposed for SDP on real world SDP data sets. Since our method produces logic rules, we present some of the insights discovered on the real world SDP data sets in Sect. 4.3. Section 2 details the existing work which is related to this study.

1.1 Main Contributions of This Study

- We experimentally compare different possible methods for combining domain costs and balancing costs in cost-sensitive software defect prediction.
- Based on our experiments, we propose a new cost-sensitive method for combining domain costs and balancing costs in software defect prediction.
- We propose a framework which calculates the balancing costs and uses our proposed cost-combining method. This framework uses recently published methods. We also modify one of these methods so that it may use multiple cost-matrices as input.
- Our experiments show that the proposed framework outperforms several existing methods when applied to the software defect prediction problem.
- We present insights into software defect prediction based on the rules discovered using our framework.

2 Related Work

2.1 Measuring Source Code

As mentioned in Sect. 1, the attributes within an SDP data set are software measures. For example, the Halstead complexity measures [5] can be used as the

attributes. There are twelve Halstead complexity measures in total. The simplest four measures are the total and distinct number of operators, and the total and distinct number of operands. From these measures, the eight other measures can be calculated. Since Halstead only counts the operators and operands, it does not consider the logic of the program. Cyclomatic complexity [11] is a measure of how many paths of execution can be taken through a program. Both Halstead's and the cyclomatic complexity measures were used to generate the publicly available NASA MDP SDP datasets [1].

2.2 Sampling Techniques

There exist several different families of techniques for treating an imbalanced data set. One highly popular family is *sampling*. Undersampling techniques remove records from the majority class, and oversampling records increase the number of majority class records. A highly popular technique for oversampling technique is *SMOTE* [3]. For each minority record, SMOTE, or *Synthetic Minority Oversampling TEchnique* chooses a random nearest neighbor. Then, a new *synthetic* record is created whose attributes' values are randomly set between the minority record and the chosen nearest neighbor. For example, if the minority record $R_m = \{10, 4, 32, 12\}$ and the random nearest neighbour is $R_{nn} = \{4, 3, 12, 15\}$, then the new synthetic record might be $R_s = \{6, 3.5, 31, 13\}$. This process can be repeated multiple times for each record based on a user defined parameter usually written as %.

There have been many extensions of SMOTE within the literature [2,6,14]. Adasyn [6] extends SMOTE by calculating the number of synthetic records that should be generated from each existing minority record. The more majority records surrounding the minority record, the more synthetic records are generated from that minority record. In the original publication, Adasyn was found to outperform SMOTE. By calculating the number of new records to generate, Adasyn eliminates the need for the % parameter.

A recent undersampling approach is *Inverse Random Undersampling* or *IRUS* [22]. The core concept of IRUS is to invert the imbalance of the data set. This is achieved by randomly removing enough records from the majority class such that the minority class becomes the majority class. A technique known as *Bagging* is then used to create an ensemble of classifiers. The author's found that IRUS outperformed several existing methods including the previously mentioned SMOTE.

2.3 Classification Methods

Decision trees are a family of classification methods such as C4.5 [13] and CSTree [10,16]. Decision trees are a data structure of nodes and branches. The initial node, known as the root node represents all records within a training data set D_T. Depending on the algorithm being used, a splitting point is calculated which partitions the records represented in the root node into several other nodes. The process is repeated for every node until a stopping criterion is met. The nodes

with no further branches coming from them are considered leaf nodes. Each branch represents the corresponding splitting condition. For example, a branch could be $A_3 > 30.5$. The node which this branch leads to will only have records for which attribute A_3's value is greater than 30.5.

The advantage of decision trees over other classification methods is that they are easy to understand by anyone. A decision tree outputs a set of logic rules which can be used to classify new records. Extracting a logic rule from a decision tree is performed by tracing a path from the root node to a leaf node and recording the branch's conditions. For example, a logic rule could look like:

IF $A_2 > 500$ AND $A_6 > 200$ AND $A_9 =$ "red" THEN POSITIVE.

The class value from the logic rule is the majority class value in the leaf node. To classify a new record, the record is tested against the conditions of each logic rule until a rule is found for which the record satisfies all conditions. The classification is then taken as the value written after "THEN". By reading the extracted logic rules, we can gain insights into the data. For example, a previous SDP study [19] extracted the following rule:

> "If a module's Halstead length is greater than 44 and the number of blank lines is less than 6 then assuming the module is defect free is the safer choice since it is 2.3 times more costly to treat it as defective."

The core difference between most decision tree algorithms is how the splitting point is calculated. For example, C4.5 calculates the *gain ratio* [13] for each possible split, then chooses the split with the highest gain ratio. In this way, C4.5 optimizes for classification accuracy. Since CSTree aims to optimize for cost, it's splitting points are calculated using *expected misclassification cost (EMC)*. EMC is calculated using two simple equations. These are the the cost of predicting all records within a data set as positive, and all as negative (Written as C_P and C_N respectively). These values are calculated as shown in Eqs. 1 and 2 where C_{TP} is the cost of a true positive, N_{TP} is the number of true positives and similarly for FP, FN, and TN.

$$C_P = N_{TP} \times C_{TP} + N_{FP} \times C_{FP} \tag{1}$$

$$C_N = N_{TN} \times C_{TN} + N_{FN} \times C_{FN}. \tag{2}$$

Decision forest algorithms such as SysFor [8] and CSForest [18,19] build multiple decision trees and classify new records by combining the individual tree classifications. SysFor is a non-cost-sensitive technique and builds multiple C4.5 trees. This is done by first finding the set of best splitting points for the original dataset, and using each as the root node split for a decision tree. The normal C4.5 procedure continues after this split. Since SysFor finds the splits based on gain ratio, the first SysFor tree is identical to the tree produced by C4.5. CSForest is similar to SysFor however it uses EMC instead of gain ratio and CSTree instead of C4.5. Therefore it is a cost-sensitive algorithm.

CSForest has been shown to outperform CSTree for the SDP problem [18,19]. However, CSForest does not account for the class imbalance problem. The

author's of CSForest also proposed an extension of CSForest called BCSForest [19] which incorporated an oversampling method into the CSForest algorithm. The experimental results demonstrated that BCSForest did not consistently outperform CSForest for the SDP problem. However since it did outperform CSForest in some cases, it demonstrated that there is room for improvement by incorporating a class imbalance treatment strategy.

2.4 Cost-Sensitive Classification for Class Imbalance Treatment

Recall from Sect. 1 the two types of costs that can be used as input for cost-sensitive classification: *domain costs* and *balancing costs*. Standoff [20] is a method for producing a classifier using balancing costs. This means that the resulting classifier is not cost-sensitive, but it is not as negatively affected by the class imbalance problem. Standoff does not use a single cost matrix for all records. Instead, it first finds the set of clusters within the minority class records and the set of clusters within the majority class records. These clusters are called class-specific-clusters [21]. A cost-matrix is then generated for each cluster. These cost-matrices are then used as input for a cost-sensitive classification algorithm. The authors of Standoff modified the cost-sensitive algorithm MetaCost [4] to use multiple cost-matrices for use in their study. Many cost-sensitive algorithms have been proposed in the literature since MetaCost. However, there has been no study on the effectiveness of Standoff when using a more recent cost-sensitive algorithm.

RBClust [21] is a fast method for finding class-specific-clusters. Since it is specifically designed for finding class-specific-clusters, it can be used in the Standoff algorithm. However, the effectiveness of doing so has not yet been studied within the literature.

3 Our Framework: BCF

We propose *Balanced Cost Framework (BCF)*, a classification framework for addressing class imbalance and cost-sensitivity. BCF is shown in Fig. 1. Within this Figure, blue rectangles represent processes, and orange ellipses represent inputs or outputs.

Standoff generates the balancing cost-matrices. However to do this, Standoff requires the class-specific-clusters to be found, and the training dataset to be supplied. In BCF, the class-specific-clusters are found from the training dataset using RBClust. Once the balancing cost-matrices are found through Standoff, BCF combines then with the user defined domain cost matrix. This cost-combining method is defined in the following section. Finally, these cost-matrices are used as input into our modified CSForest algorithm. The original CSForest algorithm was presented in the previous section.

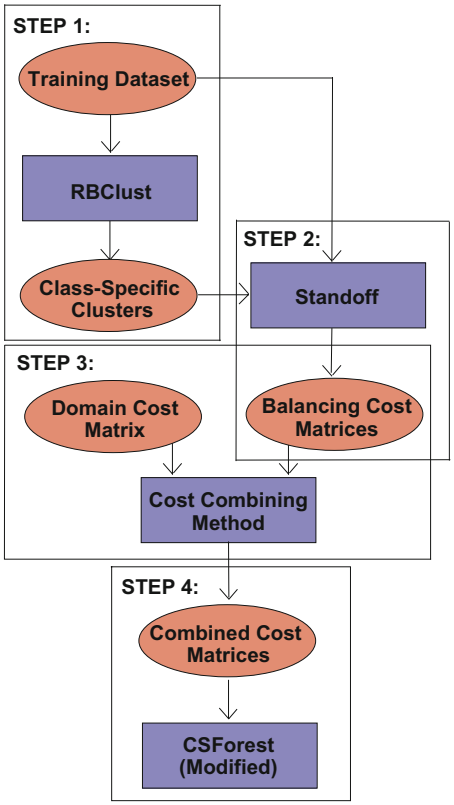

Fig. 1. The proposed framework: BCF (Color figure online)

3.1 Step 1: Generation of Class Specific Clusters (CSCs)

In Step 1, our framework uses the recent class specific clustering algorithm RBClust [21]. This step is shown in the section of Fig. 1 labeled *Step 1*. RBClust takes the training dataset as input and produces a set of class specific clusters. In addition, RBClust requires two parameters to be set. The first parameter m is a rule based classification algorithm. The second parameter θ controls the minimum size of a cluster. We use the same settings as used in the original RBClust paper [21], m = C4.5 [13] and $\theta = 0.02$. Also, as advised, we turn off pruning in C4.5 since overfitting is an advantage for RBClust.

3.2 Step 2: Calculation of Record Specific Balancing Costs

After finding the class specific clusters in Step 1, they are passed as input to the Standoff [20] algorithm. The original training dataset is also passed to Standoff. The Standoff algorithm then calculates the balancing costs for each record. We slightly modify the Standoff algorithm. Originally, the Standoff algorithm used

the balancing costs in the cost-sensitive algorithm *MetaCost* [4]. However, in our implementation, we only use Standoff to find the balancing costs. In Step 3, these costs are combined with the domain costs. Finally, in Step 4, we use a modified CSForest [18,19] instead of MetaCost for training a cost-sensitive classifier using the combined costs.

The Standoff algorithm requires the setting of three parameters, D_T - the training dataset, L a cost-sensitive algorithm, and G a clustering method. Our framework accounts for all three of the Standoff parameters. Firstly, the training dataset is passed. Since we are only using Standoff for generating the balancing cost matrices in Step 2, we do not need to specify L. Lastly, since the class-specific-clusters are found in Step 1, we do not need to specify G. It should be noted that the original Standoff [20] required a clustering algorithm to run twice, once on the majority class records, and once on the minority class records. However, RBClust does not need to separate the data into the majority and minority subsets. Therefore, by using RBClust in our framework, we eliminate the need to run the clustering algorithm twice.

3.3 Step 3: Combination of Domain Costs and Balancing Costs

In this subsection, we empirically compare different methods for combining balancing cost matrices with a domain cost matrix. In Sect. 4, we use the most reliable combination method in our proposed framework.

We will now define some notations for the domain cost matrix and the balancing cost matrices. For the domain costs, we write the TP, TN, FP and FN costs as C^{δ}_{TP}, C^{δ}_{TN}, C^{δ}_{FP}, and C^{δ}_{FN} respectively. Similarly for balancing costs, we replace δ with β. Finally, the combined costs replace δ with γ.

Two Simple Combination Methods: We now experiment two simple techniques. These techniques involve either adding or multiplying the domain costs and balancing costs. The addition technique computes the combined costs as the addition of the corresponding domain and balancing costs. For example $C^{\gamma}_{TP} = C^{\delta}_{TP} + C^{\beta}_{TP}$. The difference between the addition technique and the multiplication technique is that the corresponding costs are multiplied instead of added.

These two simple techniques are compared against a CSForest classifier. Since CSForest is a recent cost-sensitive method proposed for SDP, it acts as a reasonable baseline for performance. The data set used for evaluation is MC2′, a publicly available software defect prediction data set [1]. We measure the effectiveness of the resulting classifier in total cost. The equation for total cost is given in Eq. 3. Note that total cost is calculated using the domain costs only. We use 10-fold stratified cross validation in order to reduce variance in this analysis.

$$TotalCost = (N_{TP} \times C^{\delta}_{TP}) + (N_{FP} \times C^{\delta}_{FP}) + (N_{TN} \times C^{\delta}_{TN}) + (N_{FN} \times C^{\delta}_{FN}) \quad (3)$$

We can observe from Table 2 that addition and multiplication methods are not lowering total cost from the CSForest classifier. Thus, the results for addition and multiplication are not promising.

Table 2. MC2′ total cost comparison (Lower the better)

CSForest	Addition	Multiplication
135	136	134

Using Balancing Costs in the Prediction Phase: Typically, cost-sensitive methods use domain-costs in both the classifier training process and the classifier prediction process. In our framework, we are using CSForest which first uses the costs in training the classifier. The costs are then used in CSVoting when classifying a new record.

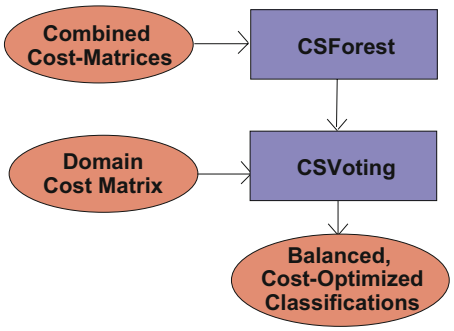

Fig. 2. Combined cost-matrices in CSForest, domain cost-matrix in CSVoting

Table 3. MC2′ total cost comparison (using only domain costs in CSVoting)

CSForest	Addition	Multiplication
135	124	132

In Table 2, the combined cost-matrices were used in both CSForest and CSVoting. However, it is possible that a better result could be achieved by only using the combined matrices once. Thus, we analyse the performance of using the combined cost-matrices when building the CSForest classifier, but only the

domain costs in CSVoting. This way, the classifications are optimized for total cost in the CSVoting step. This process is illustrated in Fig. 2.

We now compare the above approach against using the combined cost-matrices for both CSForest and CSVoting. Note that in Table 2, we used the latter approach, but in Table 3 we use the former. Thus, a comparison can be made between the two tables. We also include the total cost of a CSForest classifier for comparison.

In comparison to Table 2, the multiplication technique performs slightly better, and the addition technique is performing much better. Therefore, we decide to use only domain costs in CSVoting. Before deciding whether to use the addition or multiplication technique, we first compare with a slightly more intelligent approach.

Only Changing the FP and FN Costs: We now analyse an alternative to the two simple techniques of addition and multiplication. When generating balancing costs, it is the ratio between the FP and FN costs that is important [20]. Therefore, we decide to change only the values of FP and FN in the domain costs to generate the combined cost-matrices. This allows us to maintain the domain costs of TP and TN classifications.

To represent both the balancing cost-matrices and the domain cost-matrix for the FP and FN costs, we introduce a weight a and b for the balancing and domain cost matrix respectively. Therefore, the combined cost matrix can be written as shown in Table 4. When setting balancing cost-matrices C^{β}_{FP} can be set arbitrarily since C^{β}_{FN} is calculated as a factor of C^{β}_{FP}. In this study, we set $C^{\delta}_{FP} = 1$. When using cost-sensitive classification to deal with class imbalance, C^{β}_{FP} is typically set to 1 [20]. Therefore, in order to simplify our calculations, we set C^{γ}_{FP} to C^{δ}_{FP}, that is, both of these values are one.

Table 4. Combining only FP and FN costs

		Actual	
		Positive	Negative
Predicted	Positive	C^{δ}_{TP}	C^{δ}_{FP}
	Negative	$a \times (C^{\beta}_{FN}) + b \times (C^{\delta}_{FN})$	C^{δ}_{TN}

We experiment with applying different sets of values for weights a and b where $a + b = 1.0$. They range from 0.0 to 1.0 and with pitch width 0.1. For example, when $a = 0.5$ and $b = 0.5$, this is the same as taking the average between C^{β}_{FN} and C^{δ}_{FN} for computing C^{γ}_{FN}. Figure 3 shows a clear trend for the best combination to be at $a = 0.5$ and $b = 0.5$.

When using this approach for combining the cost-matrices, we find that we can further reduce the total cost in the MC2′ data set to 122. Finally, in comparison with the CSForest classifier, our approach generates classifications which are

approximately 10 % less costly than CSForest. Therefore, we choose to use the combined cost-matrix in Table 4 for $a = 0.5$ and $b = 0.5$ and only the domain costs in CSVoting as shown in Fig. 2. This figure also shows a second degree trend curve. This curve helps to further illustrate the trend of performance for the different weight settings.

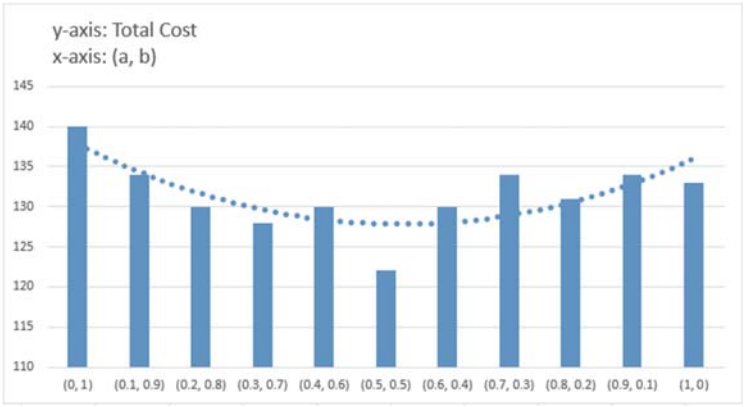

Fig. 3. Total cost comparison for different weight combinations

3.4 Step 4: Cost-Sensitive Classification Using Modified CSForest

CSForest [18,19] is a recent cost-sensitive classification method. It requires a single cost-matrix as input and thus does not handle multiple cost-matrices. This section details our modification of CSForest to use multiple cost-matrices rather than a single unchanging cost-matrix. This is done so that we can use the balancing costs generated by Standoff. Since CSForest trains a set of CSTree [10,16] decision trees, this modification is also applied to the CSTree algorithm.

A core step of CSForest is to calculate the cost of labeling an entire set of records as either negative or positive. By modifying this step to use multiple cost-matrices, this will in turn modify CSForest to use multiple cost-matrices. Each record has an associated cost-matrix. Thus, each record has a set of associated costs for TP, TN, FP, and FN classifications. Let $C^{\beta}_{TP_i}$ be the cost associated with a true positive classification for the i^{th} record from the set of records R. Similarly, we can also write TN_i, FP_i and FN_i. Also, let the set of positive class records and the set of negative class records be written as P and N respectively. The modified equation for calculating the cost of labeling a set of records as positive can be written as follows.

$$C^{\beta}_{P} = \sum_{i=1}^{|R|} \begin{cases} C^{\beta}_{TP_i} \text{ if } R_i \in P \\ C^{\beta}_{FP_i} \text{ if } R_i \in N \end{cases} \tag{4}$$

Similarly, we can write the cost of labeling a set of records as negative as follows:

$$C_N = \sum_{i=1}^{|R|} \begin{cases} C^{\beta}_{FN_i} \text{ if } R_i \in P \\ C^{\beta}_{TN_i} \text{ if } R_i \in N \end{cases} \tag{5}$$

By using the equations defined above for C_P and C_N in Eqs. 4 and 5, we are modifying CSForest to use record specific cost-matrices. Since CSForest builds a forest of CSTree decision trees, we also use Eqs. 4 and 5 when building each CSTree.

4 Experiments

In this section, we compare the effectiveness of the proposed framework against existing methods.

4.1 Experimental Setup

Compared Methods: SDP could be performed with or without addressing cost-sensitivity and/or class imbalance. Therefore we chose to compare our proposed framework with methods which address one, both or neither of these issues. This can be illustrated by a Venn diagram as shown in Fig. 4. The Venn diagram has segmented the methods into three areas, those that address class imbalance, those that address cost-sensitivity, and methods for classification. Note that the domain costs and balancing costs are inputs to cost-sensitive algorithms, but they are not methods by themselves. However, they are shown here for clarity. Figure 4 also helps to illustrate an important concept of the proposed framework. That is, it utilizes both the domain costs and the balancing costs in the classification process.

We compare our framework against seven existing methods on four of the publicly available NASA SDP [1] data sets. Recently, the cleanliness of these data sets were criticized and a data cleaning approach was proposed [17]. We use the cleaned versions of the data sets. In order to reduce variance, we have used stratified 10 cross-fold validation. All compared methods use their default parameter settings. We use the following domain costs: $C^{\delta}_{TP} = 1$, $C^{\delta}_{TN} = 0$, $C^{\delta}_{FP} = 1$ and $C^{\delta}_{FN} = 5$. This is consistent with some published studies on SDP [18,19].

4.2 Results and Discussion

The results from the experimental setup described in the previous subsection are shown in Table 5. We choose to discuss these results by closely examining several matchups described as follows:

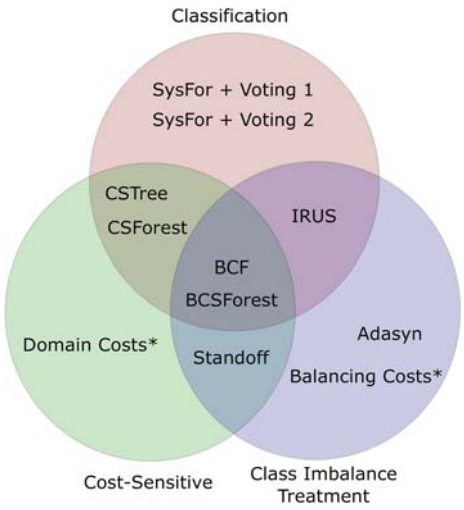

Fig. 4. Venn diagram: families of the compared methods

Table 5. Total cost comparison (Lower the better)

Data set	BCSForest	CSForest	CSTree	IRUS	Adasyn + C4.5	SysFor + Voting 1	SysFor + Voting 2	BCF
KC3′	166	174	170	157	155	156	160	**153**
MC2′	128	135	128	125	132	167	160	**123**
MW1′	128	136	173	179	147	122	130	**118**
CM1′	215	206	218	228	208	212	214	**203**
AVERAGE	159.25	162.75	172.25	172.25	160.5	164.25	166	**149.25**

Our Proposed Framework BCF vs all Other Methods: We find that in all four SDP data sets, BCF achieves the lowest total cost. On average, BCF provides an approximately 6.3 % improvement upon the next best method BCS-Forest [19]. This result suggests that BCF is a useful technique for cost-sensitive classification in SDP.

Cost-Sensitive Methods vs Non-cost-sensitive Methods: From Table 5 we can further calculate that the average performance of the cost-sensitive methods and non-cost-sensitive methods as 160.875 and 165.75 respectively. Although cost-sensitive methods provide a lower cost on average, this is not always the case. For example, in data set KC3′, the non-cost-sensitive methods achieve approximately 5.2 times lower cost than the cost-sensitive methods.

Methods Which Address Class Imbalance vs Those that Don't: Similar to the previous comparison, we can further calculate the averages of those methods that address class imbalance and those that don't. These averages are

160.3125 and 166.3125 respectively. This difference in averages demonstrates that addressing class imbalance further reduces cost on average by approximately 3.61 %.

4.3 Extracted Knowledge

The study in which CSForest was proposed [19] also provided some of the knowledge that was extracted from the logic rules which CSForest generated. In Sect. 4.2 we showed that our proposed framework outperformed CSForest. Therefore, we have chosen to also present some of the logic rules which were generated by our proposed framework. The following knowledge has been extracted from the data sets MC2′, MW1′, and KC3′. We have simply chosen the three logic rules which we think are the most interesting and understandable as follows.

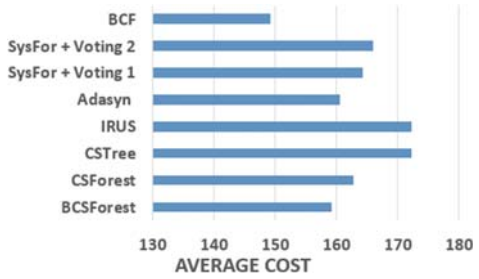

Fig. 5. Average total cost over all data sets (Lower the better)

Extracted Knowledge 1: If the number of conditions is greater than 15 and the number of blank lines is less than 11, then searching the module for bugs is 74 % less costly than assuming it contains none.

Extracted Knowledge 2: Interestingly, one rule found that out of the 17 modules in MC2′ which had Halstead Length < 41, none contained bugs. This suggests that such a module is not worth the time to search for bugs in this module. Since the cost of assuming such a module is defect-free is 0, we cannot express it as a percentage of searching the module for defects. However, we can calculate the difference between these costs. If we multiply each cost used in our SDP domain cost-matrix by 50 we can get a realistic figure. This is supported by one study which multiplied a similar set of costs by \$50 to achieve a "reasonable amount". Using Eq. 1 we can calculate that the cost of searching a module which has Halstead Length < 41 as $0 \times 0 + 1 \times 50 = \$50$. Therefore, for each such module searched for bugs, \$50 is wasted.

Extracted Knowledge 3: If a module's Halstead Effort is less than 40456.31 and the number of unique operands is less than 21 then the probability that

the module is defective is only approximately 7 % higher than defect-free. Halstead Effort can be translated into an estimate of seconds taken to program by dividing by 18 [5]. Therefore, we can extend this knowledge to the following: If a module takes less than 37 minutes to program and contains less than 21 unique operands then it should be searched for bugs since assuming it is defect-free is approximately 166 % more costly. It is interesting to note that even when there is a very small difference in probability between the module between defective, and defect-free (7 %), the cost difference is still significant (166 %). Note that trivial modules with very small Halstead Effort and number of unique operands still satisfy this rule such as getters and setters in object-oriented code.

5 Conclusion

Two issues for classification within SDP are cost-sensitivity and class imbalance. We proposed a novel framework BCF consisting of several recent methods. Our framework uses Standoff [20] to generate balancing cost-matrices for all records. It then combines them with the user-supplied domain cost matrix. We explored experimentally in Sect. 3.3 to develop an effective method for combining the balancing and domain costs. These combined cost-matrices are then used as input to a cost-sensitive classification algorithm. We modified the recent cost-sensitive method CSForest [19] to take multiple cost-matrices as input. Our experiments validate the idea that addressing cost-sensitivity in the classification can reduce the total cost, and similarly for class imbalance. Our experiments provide evidence that BCF can outperform several recent methods.

Since our framework builds decision forests, we are able to perform knowledge discovery on the studied NASA MDP data sets. In Sect. 4.3 we shared three insights which were found using our proposed framework. Our future work involves further experimental exploration to determine what factors affect the optimal method for combining balancing and domain costs.

References

1. Nasa-software defect datasets. http://openscience.us/repo/defect/mccabehalstedl. Accessed 24 July 2016
2. Bunkhumpornpat, C., Sinapiromsaran, K., Lursinsap, C.: Dbsmote: density-based synthetic minority over-sampling technique. Appl. Intell. **36**(3), 664–684 (2012)
3. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. J. Artif. Intell. Res. **16**, 321–357 (2002)
4. Domingos, P.: Metacost: A general method for making classifiers cost-sensitive. In: Proceedings of the fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 155–164. ACM (1999)
5. Halstead, M.H.: Elements of software science. North-Holland (1977)
6. He, H., Bai, Y., Garcia, E.A., Li, S.: Adasyn: adaptive synthetic sampling approach for imbalanced learning. In: 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence), pp. 1322–1328. IEEE (2008)

7. Islam, M.Z.: EXPLORE: a novel decision tree classification algorithm. In: MacKinnon, L.M. (ed.) BNCOD 2010. LNCS, vol. 6121, pp. 55–71. Springer, Heidelberg (2012). doi:10.1007/978-3-642-25704-9_7
8. Islam, Z., Giggins, H.: Knowledge discovery through sysfor: a systematically developed forest of multiple decision trees. In: Proceedings of the Ninth Australasian Data Mining Conference, vol. 121, pp. 195–204. Australian Computer Society, Inc. (2011)
9. Jiang, T., Tan, L., Kim, S.: Personalized defect prediction. In: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), pp. 279–289. IEEE (2013)
10. Ling, C.X., Sheng, V.S., Bruckhaus, T., Madhavji, N.H.: Maximum profit mining and its application in software development. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 929–934. ACM (2006)
11. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **4**, 308–320 (1976)
12. Öztürk, M.M., Cavusoglu, U., Zengin, A.: A novel defect prediction method for web pages using k-means++. Expert Syst. Appl. **42**(19), 6496–6506 (2015)
13. Quinlan, J.R.: C4. 5: programs for machine learning. Elsevier (2014)
14. Ramentol, E., Caballero, Y., Bello, R., Herrera, F.: Smote-rsb*: a hybrid preprocessing approach based on oversampling and undersampling for high imbalanced data-sets using smote and rough sets theory. Knowl. Inf. Syst. **33**(2), 245–265 (2012)
15. Ricky, M.Y., Purnomo, F., Yulianto, B.: Mobile application software defect prediction. In: 2016 IEEE Symposium on Service-Oriented System Engineering (SOSE), pp. 307–313. IEEE (2016)
16. Sheng, V.S., Gu, B., Fang, W., Wu, J.: Cost-sensitive learning for defect escalation. Knowl. Based Syst. **66**, 146–155 (2014)
17. Shepperd, M., Song, Q., Sun, Z., Mair, C.: Data quality: some comments on the nasa software defect datasets. IEEE Trans. Softw. Eng. **39**(9), 1208–1215 (2013)
18. Siers, M.J., Islam, M.Z.: Cost sensitive decision forest and voting for software defect prediction. In: Pham, D.-N., Park, S.-B. (eds.) PRICAI 2014. LNCS (LNAI), vol. 8862, pp. 929–936. Springer, Heidelberg (2014). doi:10.1007/978-3-319-13560-1_80
19. Siers, M.J., Islam, M.Z.: Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem. Inf. Syst. **51**, 62–71 (2015)
20. Siers, M.J., Islam, M.Z.: Standoff-balancing: a novel class imbalance treatment method inspired by military strategy. In: Pfahringer, B., Renz, J. (eds.) AI 2015. LNCS (LNAI), vol. 9457, pp. 517–525. Springer, Heidelberg (2015). doi:10.1007/978-3-319-26350-2_46
21. Siers, M.J., Islam, M.Z.: Rbclust: High quality class-specific clustering using rule-based classification. In: European Symposium on Artificial Neural Networks, ESANN 2016 (2016, Accepted)
22. Tahir, M.A., Kittler, J., Yan, F.: Inverse random under sampling for class imbalance problem and its application to multi-label classification. Pattern Recogn. **45**(10), 3738–3750 (2012)