Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem

Michael J. Siers*, Md Zahidul Islam

School of Computing and Mathematics, Charles Sturt University, Bathurst, NSW 2795, Australia

Abstract

Software development projects inevitably accumulate defects throughout the development process. Due to the high cost that defects can incur, careful consideration is crucial when predicting which sections of code are likely to contain defects. Classification algorithms used in machine learning can be used to create classifiers which can be used to predict defects. While traditional classification algorithms optimize for accuracy, cost-sensitive classification methods attempt to make predictions which incur the lowest classification cost. In this paper we propose a cost-sensitive classification technique called CSForest which is an ensemble of decision trees. We also propose a cost-sensitive voting technique called CSVoting in order to take advantage of the set of decision trees in minimizing the classification cost. We then investigate a potential solution to class imbalance within our decision forest algorithm. We empirically evaluate the proposed techniques comparing them with six (6) classifier algorithms on six (6) publicly available clean datasets that are commonly used in the research on software defect prediction. Our initial experimental results indicate a clear superiority of the proposed techniques over the existing ones.

© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

Software defect prediction (SDP) is the process of predicting which sections of a code are defective and which are not. Sections of software code are also referred to as modules, examples of which are functions in C/C++ programs and methods in Java programs. A module can be characterized through various measures including the number of distinct operators and operands used, the total number of operators and operands used, Difficulty, Volume Length and Cyclomatic Complexity as introduced in the Halstead [9] and McCabe [13,14] measures. Considering the measures as attributes and modules as records a dataset $D_T$ can be prepared where we have a number of records and attributes representing the previous modules for which we already know whether or not a module was defective. Therefore, in $D_T$ we also have a class attribute that labels a module as defective or non-defective. Considering $D_T$ as a training dataset a classifier can be built and applied on future modules in order to predict whether the module is defective or non-defective. This is also known as the classification task in data mining.

For the conventional classification task in data mining a classifier is generally built with the aim to minimize the number of misclassified records and thereby maximize the prediction accuracy for future records. However, in SDP a classifier is often built with the aim to minimize the...
classification cost, which is the cost associated with the classification made. That is, in SDP the classification cost is more important than the number of misclassified records. The cost of a false negative (i.e. a module being actually defective but predicted as non-defective) is generally several times higher than the cost of a false positive (i.e. a module actually being non-defective but predicted as defective). Therefore, it is often better to have several false positive prediction in order to avoid a single false negative prediction. In order to build SDP classifiers which aim to minimize the cost a cost-sensitive learning algorithm is incorporated [5,23,12,16,19]. The utilization of cost-sensitive learning in SDP can result in monetary savings for a software development group, and as such can generally be seen as more useful than SDP systems which do not utilize cost-sensitive learning.

This study extends our previous conference paper [22] which has never been published in a journal. In this paper we propose a cost-sensitive decision forest algorithm called CSForest and a suitable cost sensitive voting technique called CSVoting in order to reduce the classification cost. We also empirically compare our proposed technique with two existing cost-sensitive classifiers called Weighting [23] and CSTree [16,12] and two cost-insensitive classifiers called C4.5 [15] and SysFor [11]. We use six (6) publicly available real-world datasets available from the NASA MDP repository [20]. Our experimental results clearly indicate that the proposed cost-sensitive decision forest and cost-sensitive voting perform better than the existing techniques in minimizing the classification cost.

A problem that is often encountered in SDP is the class imbalance problem. This causes performance issues for data mining algorithms. A popular method of accounting for the class imbalance problem is oversampling. We investigate a potential technique for incorporating oversampling into CSForest. A comparison is then made with the original CSForest results in Section 6. The extensions of our conference paper which are made in this study are as follows:

- Deeper explanation of the related works.
- Extending the proposed CSForest to incorporate oversampling.
- Further experimentation.

The rest of this paper is structured as follows: in Section 2 we outline the related work. We introduce the methods CSVoting and CSForest in Section 3. We present our experimental results for CSForest and CSVoting in Section 4. We then present an idea for combatting class imbalance in CSForest in Section 5. The experimental results for combatting the class imbalance are presented in Section 6. Finally in Section 8 we present the concluding remarks and our future work.

2. Related work

In this section we introduce a number of cost-insensitive classifiers and cost-sensitive classifiers that are either somehow related to the proposed technique or empirically compared with the proposed technique in Section 4. The examples of cost-insensitive classifiers are C4.5 [15] (which is a decision tree classifier) and SysFor [11] (which is a decision forest classifier), and the examples of cost-sensitive classifiers are CSTree [16,12] and Weighting [23].

Like all other classifiers, a decision tree [15] is also built from a training dataset \( D_T \). Fig. 1 shows an example of a decision tree where the rectangles are nodes and the ovals are leaves. A node tests a non-class attribute and the edges from the node use a splitting point based on which the dataset is divided into subsets. For example, the root node of the tree (see Fig. 1) tests the attribute called Lines of Code and the splitting point is 50 based on which the dataset is divided into two subsets. The subset represented by the left side edge contains all records having the line of code \( \leq 50 \) and the subset represented by the right side edge contains all records having the line of code \( > 50 \). Note that Lines of Code is a numerical attribute. If a node tests a categorical attribute like Language then the edges split the dataset using the domain values of the attribute. For example, the left edge from the second node of the tree uses C++ and the right edge uses Java to split the dataset.

Finally a leaf node contains the records where the distribution of the class values is as homogeneous as possible for the tree. The left most leaf (Leaf 1) of the tree in Fig. 1 contains altogether eight (8) records where of which then have the class value Non-Defective and two (2) of them have the class value Defective. Therefore, any future module for which the class value (i.e. whether Defective or Non-Defective) is unknown can be predicted to be Non-Defective with a probability of \( \frac{5}{8} \) if the line of code in the module is \( \leq 50 \).

A cost-insensitive decision tree algorithm such as C4.5 [15] determines a splitting attribute (such as Lines of Code) and splitting point/s (such as 50) based on the information gain ratio of the attributes. The attribute having the highest information gain ratio is chosen as the root node. Information gain ratio of an attribute indicates the ability of the attribute to split a dataset into subsets having homogeneous distribution of class values. For example, from Fig. 1 we can see that the underlying dataset has altogether 20 records; 10 of which have the Defective class value and the other 10 have the Non-Defective class value indicating the least homogeneous distribution of class values. However, the attribute Lines of Code divides the dataset into two subsets (i.e. horizontal segments) where
Decision trees (also known as a forest) from a dataset. Random Forest [2] and SysFor [11] build an ensemble of algorithms such as Bagging [1], Boosting [6], trees have almost equal prediction accuracy. Therefore, a decision trees from the same dataset where all of these that different decision tree algorithms often build different by a single decision tree. It is also evident from the fact contains more rules/patterns than the set of rules obtained (a decision tree reaches to a leaf node. The same approach is then applied recursively on each subset compared to the immediate parent subset. Otherwise improvement of the distribution of class values improves in both subset there are 8 records having Defective and 4 records having Non-Defective class values. Therefore, the homogeneity of the distribution of class values improves. However, it is likely that a dataset contains trees which disagree with one another while classifying R. Therefore, in order to classify R a voting method is used. For example, Random Forest [2] uses the most common classification for R by all trees Ti as the final classification.

SysFor uses either of the two voting algorithms: Voting1 or Voting2. To classify an unlabelled record R, Voting1 finds the set of all leaves L100 where R falls into that satisfy two conditions. Condition 1: all the records in the leaf contain a single class value. Condition 2: the number of records is greater than a user defined threshold, or greater than \(|D|/|N| \times C\) where |D| is the total number of records in the dataset, |N| is the number of leaves in the tree, and C is a user defined constant. Considering the L100 leaves, a sum of the supports of each class value is then computed. The class value with the highest sum is taken as the final classification of R. If the number of L100 leaves is zero, then all leaves are considered instead. Voting2 works by finding the set of all leaves that an unlabeled record R falls into L and uses the prediction of the leaf in L with the highest accuracy: \(L_{\text{max}}\).

The classifiers introduced above are cost-insensitive in the sense that they only aim to increase the prediction/classification accuracy instead of minimizing the classification cost. There are mainly two types of cost-sensitive methods; the wrapper methods and direct methods. Wrapper methods use an existing cost-insensitive method as a base classifier and affect the classification process to make the resulting predictions consider cost. Direct methods consider costs directly in the classifier building process.

There are many wrapper methods [23,5,19] used for cost-sensitive classification. The Weighting method [23] applies a decision tree algorithm on each \(D_i\) and thereby builds a decision tree \(T_i\). Therefore, from the \(n\) datasets it builds \(n\) decision trees. SysFor [11] builds the first tree \(T_1\) from \(D_1\) by using a conventional decision tree algorithm such as C4.5. The attribute having the maximum information gain ratio is chosen by C4.5 as the test attribute for the root node of \(T_1\). The attribute having the 2nd best information gain ratio is then chosen as the root node of the second tree \(T_2\). Once the root node is chosen for \(T_2\) all other subsequent nodes are chosen as usual following the C4.5 algorithm. This process of choosing a new attribute at the root node continues as long as the information gain ratio of the chosen attribute is within a user defined threshold of the gain ratio of the best attribute chosen in \(T_1\). If more trees are needed to be generated then the same process is applied on the nodes of the next level instead of the root node.

### Table 1
Cost metric.

<table>
<thead>
<tr>
<th>Prediction</th>
<th>Cost</th>
<th>Action taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP (true positive)</td>
<td>1</td>
<td>The module is fixed</td>
</tr>
<tr>
<td>TN (true negative)</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>FP (false positive)</td>
<td>1</td>
<td>The module is attempted to be fixed, but discovered to be defect free</td>
</tr>
<tr>
<td>FN (false negative)</td>
<td>5</td>
<td>The defective module goes undetected and causes costly issues later in the software development process</td>
</tr>
</tbody>
</table>

### Table 2
Total classification cost of the existing methods.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>C4.5</th>
<th>SVM + Voting1</th>
<th>SysFor + Voting1</th>
<th>C4.5 +CSTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC2’</td>
<td>154</td>
<td>162</td>
<td>164</td>
<td>171</td>
</tr>
<tr>
<td>PC1’</td>
<td>287</td>
<td>275</td>
<td>255</td>
<td>257</td>
</tr>
<tr>
<td>KC1’</td>
<td>1282</td>
<td>1476</td>
<td>1404</td>
<td>1397</td>
</tr>
<tr>
<td>PC3’</td>
<td>640</td>
<td>623</td>
<td>664</td>
<td>663</td>
</tr>
<tr>
<td>MC1’</td>
<td>272</td>
<td>321</td>
<td>275</td>
<td>264</td>
</tr>
<tr>
<td>PC2’</td>
<td>85</td>
<td>82</td>
<td>81</td>
<td>86</td>
</tr>
</tbody>
</table>

### Table 3
Impact of the proposed CSVoting on SysFor.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>SysFor + Voting1</th>
<th>SysFor + Voting2</th>
<th>SysFor + CSVoting</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC2’</td>
<td>164</td>
<td>171</td>
<td>145</td>
</tr>
<tr>
<td>PC1’</td>
<td>255</td>
<td>257</td>
<td>271</td>
</tr>
<tr>
<td>KC1’</td>
<td>1404</td>
<td>1397</td>
<td>1268</td>
</tr>
<tr>
<td>PC3’</td>
<td>664</td>
<td>663</td>
<td>571</td>
</tr>
<tr>
<td>MC1’</td>
<td>275</td>
<td>264</td>
<td>270</td>
</tr>
<tr>
<td>PC2’</td>
<td>81</td>
<td>86</td>
<td>81</td>
</tr>
</tbody>
</table>

Lines of Code, Impact of the proposed CSVoting on SysFor.

### Table 2
Total classification cost of the existing methods.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>C4.5</th>
<th>SVM + Voting1</th>
<th>SysFor + Voting1</th>
<th>C4.5 +CSTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC2’</td>
<td>154</td>
<td>162</td>
<td>164</td>
<td>171</td>
</tr>
<tr>
<td>PC1’</td>
<td>287</td>
<td>275</td>
<td>255</td>
<td>257</td>
</tr>
<tr>
<td>KC1’</td>
<td>1282</td>
<td>1476</td>
<td>1404</td>
<td>1397</td>
</tr>
<tr>
<td>PC3’</td>
<td>640</td>
<td>623</td>
<td>664</td>
<td>663</td>
</tr>
<tr>
<td>MC1’</td>
<td>272</td>
<td>321</td>
<td>275</td>
<td>264</td>
</tr>
<tr>
<td>PC2’</td>
<td>85</td>
<td>82</td>
<td>81</td>
<td>86</td>
</tr>
</tbody>
</table>

The same approach is then applied recursively on each subset separately in order to find the next splitting attribute and splitting point. This process of dividing a dataset into subsets continues as long as there is an attribute that can improve the homogeneity of the class distribution in the child subsets compared to the immediate parent subset. Otherwise a decision tree reaches to a leaf node.

A single decision tree obtains a set of rules/patterns from a data set. An example of a rule can be Lines of Code \(\leq 50 \Rightarrow\) Non-Defective. However, it is likely that a dataset contains more rules/patterns than the set of rules obtained by a single decision tree. It is also evident from the fact that different decision tree algorithms often build different decision trees from the same dataset where all of these trees have almost equal prediction accuracy. Therefore, a number of algorithms such as Bagging [1], Boosting [6], Random Forest [2] and SysFor [11] build an ensemble of decision trees (also known as a forest) from a dataset.

Bagging [1] first generates a number of datasets \(D_i\) \((i = 1, 2, \ldots, n)\) from a training dataset \(D_T\) by randomly selecting a set of records for each \(D_i\) from \(D_T\). It then

\[ R = \frac{1}{N} \sum_{i=1}^{n} R_i 

\]
terms and notations as follows. Before discussing the equations we introduce a set of attributes \( A_i \) and total number of records in the class values, number of records having the class values carefully computed considering the classification cost of a False Positive prediction. The new class distribution is changed to 15 defective records and (we call them non-defective records), Weighting may CSTree [12,16] uses expected classification cost reduction expected classification cost CSTree uses Eqs.(1) and (2). In order to calculate \( C_{STree} \) first calculates the expected total classification cost change the class distribution to 15 defective records and same way as C4.5. If \( A_i \) is a numerical attribute then \( D_T \) is divided into two subsets using the best splitting point. If \( A_i \) is a categorical attribute with \( k \) different values (i.e. domain size \( A_i \) is \( k \)) then \( D_T \) is divided into \( k \) subsets. \( C_P \) and \( C_N \) are the costs of labeling the records within the \( i \)th subset as positive and negative, respectively. The expected classification cost for \( A_i \) is then computed using the following equation:

\[
E_{A_i} = 2 \times \sum_{i=1}^{k} \frac{C_P \times C_N^i}{C_P + C_N^i}
\]

Finally, the expected classification cost reduction for \( A_i \) is given by \( E - E_{A_i} - T_C \), where \( T_C \) is the total test cost for all examples on \( A_i \). While searching for a splitting attribute \( A_i \), CSTree iterates over all possible splitting attributes \( A_i \in A; \forall i \) and chooses the one with the highest expected cost reduction. This splitting attribute needs to satisfy \( E - E_{A_i} - T_C > 0 \), else no splitting attribute is found. Similar to C4.5, this process of choosing a splitting attribute recursively continues for every subset. CSTree also implements post-pruning similar to the C4.5 decision tree algorithm but is instead guided by total classification cost.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>C4.5</th>
<th>SVM</th>
<th>SysFor + Voting1</th>
<th>SysFor + Voting2</th>
<th>CSC + C4.5</th>
<th>CTree</th>
<th>CSForest + CVSampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC2</td>
<td>154</td>
<td>162</td>
<td>164</td>
<td>171</td>
<td>154</td>
<td>165</td>
<td>129</td>
</tr>
<tr>
<td>PC1</td>
<td>287</td>
<td>275</td>
<td>255</td>
<td>257</td>
<td>283</td>
<td>290</td>
<td>276</td>
</tr>
<tr>
<td>KC1</td>
<td>1282</td>
<td>1476</td>
<td>1404</td>
<td>1397</td>
<td>1226</td>
<td>1187</td>
<td>1168</td>
</tr>
<tr>
<td>PC3</td>
<td>640</td>
<td>623</td>
<td>664</td>
<td>663</td>
<td>637</td>
<td>586</td>
<td>521</td>
</tr>
<tr>
<td>MC1</td>
<td>272</td>
<td>321</td>
<td>275</td>
<td>264</td>
<td>272</td>
<td>291</td>
<td>261</td>
</tr>
<tr>
<td>PC2</td>
<td>85</td>
<td>82</td>
<td>81</td>
<td>86</td>
<td>83</td>
<td>84</td>
<td>80</td>
</tr>
</tbody>
</table>

There are four types of predictions:

- True positive (TP): The module is correctly predicted as defective.
- True negative (TN): The module is correctly predicted as non-defective.
- False positive (FP): The module is incorrectly predicted as defective.
- False negative (FN): The module is incorrectly predicted as non-defective.

\( N_{TP} \) is the number of TP predictions, \( N_{FP} \) is the number of FP predictions, and so on. The cost associated with a TP prediction is \( C_{TP} \) and likewise the cost of a FP prediction is \( C_{FP} \), and so on.

In order to find the test attribute at the root node CSTree first calculates the expected total classification cost of the whole dataset using Eq. (1) that uses the cost of labeling a set of records as positive \( C_P = N_{TP} \times C_{TP} + N_{FP} \times C_{FP} \) and the cost of labeling a set of records as negative \( C_N = N_{TN} \times C_{TN} + N_{FN} \times C_{FN} \):

\[
E = \frac{2 \times C_P \times C_N}{C_P + C_N}
\]

CSTree then computes the ability of each attribute \( A_i \in A \) (where \( A \) is the set of attributes in \( D_T \)) to reduce the classification cost. Finally the attribute with the highest classification cost reduction is taken as the root attribute. In order to compute the ability of \( A_i \) to reduce the classification cost CSTree divides \( D_T \) into mutually exclusive horizontal segments (i.e. subsets) using \( A_i \) in the same way as C4.5. If \( A_i \) is a numerical attribute then \( D_T \) is divided into two subsets using the best splitting point. If \( A_i \) is a categorical attribute with \( k \) different values (i.e. domain size \( A_i \) is \( k \)) then \( D_T \) is divided into \( k \) subsets. \( C_P \) and \( C_N \) are the costs of labeling the records within the \( i \)th subset as positive and negative, respectively. The expected classification cost for \( A_i \) is then computed using the following equation:

\[
E_{A_i} = 2 \times \sum_{i=1}^{k} \frac{C_P \times C_N^i}{C_P + C_N^i}
\]
Table 5

<table>
<thead>
<tr>
<th>Dataset</th>
<th>C4.5</th>
<th>SVM</th>
<th>CSForest + CSVoting</th>
<th>CSTree</th>
<th>CSForest</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC1</td>
<td>43</td>
<td>30</td>
<td>25</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>MC2</td>
<td>154</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>KC1</td>
<td>108</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>KC2</td>
<td>207</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>PC1</td>
<td>439</td>
<td>121</td>
<td>426</td>
<td>426</td>
<td>426</td>
</tr>
<tr>
<td>PC2</td>
<td>80</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>PC3</td>
<td>80</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
</tbody>
</table>

For each attribute, the new synthetic record takes a value between the attribute values of the current example and of the nearest neighbour. SMOTE uses a user defined parameter which decides how many new records to create. A value of 300 will create 300% more examples.

Safe-Level-Smote [3] defines a formula for the safe level of an example based on the number of positive examples within its k nearest neighbours. The safe level is then calculated for each positive example and a random nearest neighbour. The safe level ratio is then computed as the safe level of the original positive instance divided by the safe level of the random nearest neighbour. If the safe level of the nearest neighbour is equal to the safe level of the original positive example, a new example is computed similar to SMOTE. However, if the nearest neighbour has a higher safe level than the original example, a new synthetic record is created closer to the nearest neighbour. Likewise, if the safe level of the original example is higher, a new synthetic example is created closer to the original example. In the case where both safe levels are 0, no new example is created. In the case where the safe level ratio is infinity and the safe level of the original example is not 0, the original record is duplicated.

3. Our method

In this study we propose a cost-sensitive voting called CSVoting (Cost-Sensitive Voting) for a decision forest, and a cost-sensitive decision forest algorithm called CSForest. Since we consider the scenario for Software Defect Prediction (SDP) we focus on two class classification; defective and non-defective. However, our methods can easily be extended for multi-class cost-sensitive classification.

CSVoting classifies a record \( R_i \) as follows. A record \( R_i \) falls in (i.e. satisfies the logic rule of) a single leaf of a tree \( T_j \) of a forest \( F \). Therefore, \( R_i \) falls in as many leaves as the number of trees in \( F \). Let us consider that \( F \) has \( n \) trees and therefore \( R_i \) falls in \( n \) leaves \( L_1, L_2, \ldots, L_n \). For each leaf \( L_j \) CSVoting calculates the cost of labeling the records belonging to \( L_j \) as positive \( C_p^j \) (i.e. the positive classification cost) and the cost of labeling them as negative \( C_n^j \) (i.e. the negative classification cost). It then computes the total positive classification cost for all \( n \) leaves as \( C_p = \sum_{j=1}^{n} C_p^j \) and total negative classification cost as \( C_n = \sum_{j=1}^{n} C_n^j \). Finally, \( R_i \) is classified as positive if \( C_p < C_n \), otherwise it classified as negative.

The proposed CSForest modifies SysFor [11] by using the Classification Cost Reduction (CCR) \( E - E_A - T^+_C \) (see Eqs. (1) and (2)) instead of the Gain Ratio as the splitting criterion. CSForest also uses a cost-sensitive pruning instead of the pruning used by C4.5 [15]. As we know C4.5 and SysFor prune a tree if the expected number of misclassification for future records does not increase significantly due to the pruning. However, CSForest prunes a tree if the expected classification cost does not increase significantly due to the pruning. Additionally, unlike CSTree CSForest allows a tree to first fully grow and then get pruned. CSTree uses the splitting criterion \( E - E_A - T^+_C > 0 \) whereas CSForest modifies the criterion to be \( E - E_A - T^+_C \geq 0 \) and thereby builds a deeper tree than the one built by CSTree, before pruning.
While CSTree builds a single tree CSForest builds an ensemble of tree and then uses CSVoting for the classification of records/modules. Algorithm 1 presents the CSForest algorithm in details. The CSForest algorithm takes as input a dataset $D_T$ and a number of user defined values on the number of records $T$, goodness threshold $c$, separation threshold $c$, pruning confidence factor $\tau$, and minimum number of records in a leaf $m_l$.

Algorithm 1. CSForest.

```
Input: $D_T$, $T$, $c$, $c$, $m_l$
Output: a set of trees $F$
1 initialise a set of decision trees $F$ to null;
2 initialise a set of good attributes $A^g$ to null;
3 initialise a set of split points $P^s$ to null;
4 set $A^f$, and $P^f$ by calling GetGoodAttributes($D_T$, $c$, $c$);
5 initialise $i$ to 1;
6 while $|F| < T$ and $|F| < |A|$ do
7     $T_j = \text{BuildCSTree}(D_T, A^j, P_j, c, m_l)$;
8     $F = F \cup T_j$;
9     $i = i + 1$;
10    end
11
12 initialise $K$ to $|F|$;
13 while $|F| < T$ and $i < K$ do
14    /* divide the data set $D_T$ */
15    if $A_j$ is categorical then
16        $D_T = \{D_1, D_2, \ldots, D_{A_j}\}$;
17    end
18    if $A_j$ is numerical then
19        $D_T = \{D_1, D_2\}$;
20    end
21    /* divide $D_T$ with $A^j$ */
22    for $j = 1$ to $|D_T|$ do
23        initialise $A^f_j$, and $P^f_j$ to null;
24        /* get attributes $A^f_j$, and $P^f_j$ */
25        $A^f_j$, and $P^f_j$ = GetGoodAttributes($D_i$, $r$, $c$);
26    end
27    initialise number of possible trees, $t_p$ to null;
28    calculate, $t_p = \sum_{j=1}^{\mid D_T \mid} p^{f_j \mid 0}$;
29    initialise $x$ to 1;
30    while $|F| < T$ and $x \leq t_p$ do
31        for $j = 1$ to $|D_T|$ do
32            if $A^f_j > x$ then
33                $T_j = \text{BuildCSTree}(D_i, a_{j+1} \in A^f_j, p_{j+1} \in P^f_j, c, m_l)$;
34            end
35        end
36        if $A^f_j \leq x$ then
37                $T_j = \text{BuildCSTree}(D_i, a_j \in A^f_j, p_j \in P^f_j, c, m_l)$;
38        end
39        Build a tree $T_{new}$ by joining the root node of each $T_j(1 \leq j \leq |D_T|)$ as a child node with the root node of $T_i$;
40        $F = F \cup T_{new}$;
41        $x = x + 1$;
42    end
43    $i = i + 1$;
44    end
45 return $F$;
```

Using GetGoodAttributes($D_T$, $c$, $c$) CSForest first computes the Classification Cost Reduction (CCR) ability of each attribute $A_j \in A^f$. The set $P^f$ stores the corresponding splitting point/s $P_i$ for $A_j \in A^f$. Note that if $A_j$ is a numerical attribute
it can be selected more than once within \( A^g \) based on its
different splitting points provided a splitting point of \( A_i \) is
at least \( \epsilon \) (a user defined separation threshold) distant
from all other splitting points of \( A_i \).

Once CSForest has \( A^g \) and \( P^g \), it builds a cost sensitive
tree using each of the splitting attributes and splitting
points at the root node, as shown in Line 7 of Algorithm 1.
Each cost sensitive tree uses the pruning confidence factor
c and minimum number of leaves \( n_l \) in a leaf, a splitting
attribute \( A_i \) and the corresponding splitting point \( P_i \). The
tree \( T_i \) having the attribute \( A_i \) at the root node is then
added in the set of trees \( F \), as long as \( j(F) < T \) and \( A^g < T \).

If the number of trees created up to Line 10 of
Algorithm 1 is less than the user defined number of trees
(i.e. \( |F| < T \)) then CSForest creates more trees by using the
same approach in Level 2 of the trees built so far. For
example, if a user asks for 20 trees and CSForest has built
only 10 trees so far (up to Line 10) then it builds more
trees. The 11th tree is built considering the root node of
the 1st tree and dividing the dataset \( DT \) into the subsets
\( DT = \{ D_1, D_2, \ldots, D_{|A_i|} \} \) based on \( A_1 \). For each subset \( D_i \) it
applies the same approach to find the set of good attri-
butes (Line 22) and builds a tree from the segment (Line
30 and Line 33). The root nodes of these trees are then
joined with the original root node \( A_1 \) (see Line 27 to Line
39) and thus the 11th tree is built.

The maximum number of trees that can be built from a
root node \( A_i \) \( A^g \) is computed using the number of good
attributes in each subset \( D_j \) \( DT \) and the number of records
in the subset, as shown in Line 25 of Algorithm 1. Once all
possible trees are built from \( A_1 \) CSForest uses the same
approach for \( A_2 \) \( A^g \) and so on until it creates the \( T \) trees.

4. Experimental results

Following the common trend in Software Defect Pre-
prediction (SDP) we in our experiments also consider that a
false negative classification is several times (in this
instance 5 times) more costly than a true positive classi-
fication. We consider attempting to fix a non-defective
module to be similar in cost to detecting and fixing a
defective module. Our cost metric is shown in Table 1.
Since the value of \( T^c_i \) is not clearly defined in the literature
[16] and has been considered to be 0 [12], we also assume
\( T^c_i = 0 \). We use the implementation of Weighting as avail-
able in WEKA [10] and called Cost Sensitive Classifier
(CSC). Also used from WEKA is the SMO [17,18] method
of creating a SVM.

Table 6
Weighted precision comparison.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>C4.5</th>
<th>SVM</th>
<th>SysFor + Voting1</th>
<th>SysFor + Voting2</th>
<th>CSC + C4.5</th>
<th>CSTree</th>
<th>CSForest + CSVoting</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC2’</td>
<td>0.538</td>
<td>0.563</td>
<td>0.487</td>
<td>0.472</td>
<td>0.538</td>
<td>0.383</td>
<td>0.446</td>
</tr>
<tr>
<td>KC1’</td>
<td>0.297</td>
<td>0.44</td>
<td>0.486</td>
<td>0.5</td>
<td>0.316</td>
<td>0.24</td>
<td>0.272</td>
</tr>
<tr>
<td>PC3’</td>
<td>0.516</td>
<td>0.652</td>
<td>0.625</td>
<td>0.588</td>
<td>0.474</td>
<td>0.366</td>
<td>0.455</td>
</tr>
<tr>
<td>PC3’</td>
<td>0.309</td>
<td>0.521</td>
<td>0.297</td>
<td>0.301</td>
<td>0.313</td>
<td>0.337</td>
<td>0.392</td>
</tr>
<tr>
<td>MC1’</td>
<td>0.818</td>
<td>0.381</td>
<td>0.85</td>
<td>0.724</td>
<td>0.818</td>
<td>0.472</td>
<td>0.639</td>
</tr>
<tr>
<td>PC2’</td>
<td>0</td>
<td>0.143</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.111</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7
Weighted recall comparison.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>C4.5</th>
<th>SVM</th>
<th>SysFor + Voting1</th>
<th>SysFor + Voting2</th>
<th>CSC + C4.5</th>
<th>CSTree</th>
<th>CSForest + CSVoting</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC2’</td>
<td>0.154</td>
<td>0.122</td>
<td>0.132</td>
<td>0.112</td>
<td>0.154</td>
<td>0.18</td>
<td>0.375</td>
</tr>
<tr>
<td>KC1’</td>
<td>0.042</td>
<td>0.042</td>
<td>0.071</td>
<td>0.066</td>
<td>0.047</td>
<td>0.077</td>
<td>0.101</td>
</tr>
<tr>
<td>PC3’</td>
<td>0.095</td>
<td>0.03</td>
<td>0.048</td>
<td>0.051</td>
<td>0.129</td>
<td>0.226</td>
<td>0.168</td>
</tr>
<tr>
<td>PC3’</td>
<td>0.06</td>
<td>0.042</td>
<td>0.036</td>
<td>0.036</td>
<td>0.065</td>
<td>0.118</td>
<td>0.179</td>
</tr>
<tr>
<td>MC1’</td>
<td>0.067</td>
<td>0.026</td>
<td>0.063</td>
<td>0.082</td>
<td>0.067</td>
<td>0.063</td>
<td>0.093</td>
</tr>
<tr>
<td>PC2’</td>
<td>0</td>
<td>0.013</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.013</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8
Incorporating oversampling into CSForest.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>CSForest Setup 1</th>
<th>Setup 2</th>
<th>Setup 3</th>
<th>Setup 4</th>
<th>Setup 5</th>
<th>Setup 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC2’</td>
<td>129</td>
<td>136</td>
<td>114</td>
<td>137</td>
<td>132</td>
<td>135</td>
</tr>
<tr>
<td>PC’1</td>
<td>276</td>
<td>250</td>
<td>256</td>
<td>267</td>
<td>271</td>
<td>311</td>
</tr>
<tr>
<td>KC1’</td>
<td>1168</td>
<td>1206</td>
<td>1200</td>
<td>1212</td>
<td>1206</td>
<td>1228</td>
</tr>
<tr>
<td>PC’3</td>
<td>521</td>
<td>555</td>
<td>561</td>
<td>647</td>
<td>607</td>
<td>576</td>
</tr>
<tr>
<td>MC1’</td>
<td>261</td>
<td>250</td>
<td>264</td>
<td>269</td>
<td>279</td>
<td>247</td>
</tr>
<tr>
<td>PC’2</td>
<td>80</td>
<td>88</td>
<td>80</td>
<td>87</td>
<td>83</td>
<td>88</td>
</tr>
</tbody>
</table>

Following the common trend in Software Defect Pred-
prediction (SDP) we in our experiments also consider that a
false negative classification is several times (in this
instance 5 times) more costly than a true positive classi-
fication. We consider attempting to fix a non-defective
module to be similar in cost to detecting and fixing a
defective module. Our cost metric is shown in Table 1.
Since the value of \( T^c_i \) is not clearly defined in the literature
[16] and has been considered to be 0 [12], we also assume
\( T^c_i = 0 \). We use the implementation of Weighting as avail-
able in WEKA [10] and called Cost Sensitive Classifier
(CSC). Also used from WEKA is the SMO [17,18] method
of creating a SVM.
For the experimentation we use six (6) popular software-defect-prediction datasets namely MC2', PC1', KC1', PC3', MC1' and PC2' that are publicly available from NASA MDP [20]. The original MDP datasets were criticized to be unclean having incorrect values [21,8,7]. As a result NASA responded to the criticism by releasing cleaned versions of the datasets. In this study we use only the clean datasets.

### 4.1. Cost comparison of the classifiers

The existing methods studied in our experiments are SysFor with Voting 1, SysFor with Voting 2, Weighting (called CSC in this section), CSTree, and C4.5. As CSC is a wrapper method, we need to choose a base classifier. Since SysFor and CSTree are closely related to C4.5, and C4.5 is used in this experimental analysis, we choose C4.5 as the base classifier for CSC. Upon making the comparison between the five existing methods used in this study, we observe in Table 2 that there is no clear winner when the goal is achieving low classification cost. The highest performing results are indicated by bold font in the tables.

We realize that SysFor is a cost-insensitive decision forest which also uses one of the two cost-insensitive voting techniques. We next investigate the effectiveness of SysFor in reducing the classification cost when we replace its original voting by our proposed cost-sensitive voting called CSVoting. We can see in Table 3 that the proposed CSVoting achieves the minimum classification cost in four out of six datasets. Therefore, CSVoting appears to be promising and we next use it with the proposed CSForest algorithm.

We now evaluate the proposed CSForest and CSVoting algorithms by combining them together and then comparing their classification cost with six (6) existing techniques on six (6) datasets as shown in Table 5. The proposed technique achieves the lowest total classification cost in out of six (6) datasets. While comparing Table 5 with Table 2 we can see the appearance of a clear winner in Table 4, which is the proposed technique. The results in Table 5 reveal the effectiveness of the proposed technique suggesting the usefulness of the modifications such as using a cost-sensitive forest with a cost-sensitive voting method and using the fully grown cost-sensitive trees before pruning them back.

The classification cost of each fold is also included in Table 5 for a more detailed comparison.

#### 4.2. Further analysis using weighted precision and weighted recall

Precision and recall are commonly used for comparing the performance of different classification algorithms. Precision represents a model’s ability to correctly make a positive prediction. The formula for precision is given in Eq. (3). Recall represents a model’s ability to identify the positive cases. The formula for recall is given in Eq. (4):

\[
\text{Precision} = \frac{N_{TP}}{N_{TP} + N_{FP}} \tag{3}
\]

\[
\text{Recall} = \frac{N_{TP}}{N_{TP} + N_{FN}} \tag{4}
\]

Due to the cost-sensitive context of this study, we introduce $C_{TP}$, $C_{FP}$, $C_{FN}$, and $C_{TN}$ into the calculations of precision and recall. Thus, weighted versions of precision and recall are given by the following equations, respectively,

\[
\text{Weighted Precision} = \frac{N_{TP} \times C_{TP}}{N_{TP} \times C_{TP} + N_{FP} \times C_{FP}} \tag{5}
\]

\[
\text{Weighted Recall} = \frac{N_{TP} \times C_{TP}}{N_{TP} \times C_{TP} + N_{FN} \times C_{FN}} \tag{6}
\]

Note that since $C_{TP}$, $C_{FP}$ are both equal to 1 in this study, weighted precision is equal to precision. However, weighted recall gives a different value to recall since $C_{TP} = 5$. The cost-insensitive methods used in this study (C4.5, SysFor, SVM) are designed to make high accuracy predictions. Cost-sensitive methods achieve lower classification cost than cost-insensitive methods by being willing to compromise accuracy if necessary. Thus, we can expect cost-insensitive methods to attain higher precision than cost-sensitive methods. FN has a high cost associated with it in this study. Therefore, the cost-sensitive methods used have focused more heavily on reducing $N_{FN}$ than the cost-insensitive methods. We can expect to see a higher performance in weighted recall when using cost-sensitive methods since $C_{FN}$ is used in calculating weighted recall.

Table 6 shows that the cost-insensitive methods SVM and SysFor are the highest performing methods in terms of weighted precision. However, since weighted precision is still the same as precision, it is not an optimal way to compare the methods in a cost-sensitive context.

![Fig. 2. Incorporating oversampling into CSForest.](image-url)
In Table 7 we observe a clear superiority of the proposed technique over the existing techniques. In Section 4.1 we concluded that CSForest coupled with CSVoting produced the lowest cost predictions than the existing techniques (see Table 4). Since weighted recall places a heavy emphasis on minimizing $N_{fn}$, CSForest with CSVoting also produces the greatest weighted recall in four out of six datasets (see Table 7). In the cost-sensitive context, minimising the prediction cost is the main goal. Therefore we consider Table 4 to provide the most useful empirical result.

5. Potential solution to class imbalance

It is well known that software defect prediction datasets typically suffer from the class imbalance problem where the datasets contain only few records with defective modules compared to the number of defect-free modules. For example, the MC’1 and PC’2 datasets contain only 0.73% and 1.01% defective modules. The ability of a classifier to predict the records with the minority class value (such as the defective module) can be challenged by the class imbalance problem in a dataset.

We investigate the incorporation of oversampling techniques into the decision forest building process in order to further reduce cost in SDP. In order to incorporate oversampling techniques, we use SMOTE [4] and Safe-Level-Smote [3]. However, we do not use SMOTE or Safe-Level-Smote once on the original data set in order to create one oversampled data set. The process we use is illustrated in Fig. 2. We use SMOTE or Safe-Level-Smote $x$ times on the original data set to create $x$ new versions of the original data set. These data sets can be seen in Fig. 2 as $D_1$, $D_2$, and $D_x$. From each of the oversampled data sets, we build a CSForest; $F_1$, $F_2$, and $F_x$. Once the $x$ CSForests have been built, the trees from each CSForest are collected together into a single CSForest. In order to compare with the previous results which used 20 trees, we experiment with both 2 resampled versions of the dataset with a forest of 10 trees built from each, and with 5 resampled versions of the dataset with a forest of 4 trees built from each. In addition to experimenting with SMOTE and Safe-Level-Smote we also experimented by adding very small amounts of noise to each defective record in order to create new records. To add small amounts of noise, we copied each minority class record four times. For each of the copies, we chose a random attribute and randomly decreased or increased the value by a small amount.

6. Class imbalance experiments

We compare the cost of the original CSForest with a modified version of CSForest that incorporates an oversampling technique as described in the previous section. We use SLS to refer to Safe-Level-Smote. The number in parenthesis in the following table refers to how many oversampled versions of the original data set were created. To make references easier we refer to the columns which represent CSForests which incorporate class imbalance as different setups. For example, Setup 2 is a CSForest which was built using five oversampled versions of the original data set using SMOTE. Setup 3 is a CSForest which was built using two oversampled versions of the original data set using Safe-Level-Smote. Similarly, Setup 5 is a CSForest which was built using two oversampled versions of the original data set by adding noise as described in the previous section.

In Table 8 we find that in 4 of the datasets there is a combination that can achieve a cost at least as low as the original CSForest. The other two datasets, KC1’ and PC3’, were the only two datasets with a number of defective examples greater than 100 as shown in Table 9. Perhaps these two datasets have a sufficient number of defective examples such that the defective class is not under-represented. Thus, it appears that the use of oversampling techniques can give a better result where the number of defective examples is less than 100. We can observe that adding noise is generally ineffective as a resampling technique but still manages to achieve lowest cost in MC1’. In the twelve cases in which Safe-Level-Smote was incorporated, only one managed a lower cost.

7. Examples of knowledge discovery by our cost-sensitive forest

A subject of our future research is the knowledge extraction from decision forests built by the CSForest method. However, for the interested reader, we present a few takeaway insights discovered by the proposed cost-sensitive forest. Since these insights are described using attributes from the datasets, a few attributes are first explained in detail in the following subsection.

7.1. Related attributes

Rather than count the total number of lines of executable code in a module, the logical line count may be used. The logical line count will only count a statement which has been split into multiple lines as one line. Cyclomatic complexity [13] is the number of paths of execution that may be taken through a module. Like all of the Halstead complexity measures [9], Halstead difficulty and Halstead length are calculated using both the total counts, and the number of distinct operators and operands. The formulae for cyclomatic complexity density [14], Halstead difficulty, and Halstead length are provided in the following equations, respectively.

\[
\text{Cyclomatic Complexity Density} = \frac{\text{Cyclomatic Complexity}}{\text{Logical Line Count}}
\]  
(7)

\[
\text{Halstead Difficulty} = \frac{\text{Number of Distinct Operators}}{\text{Total Number of Operators}} \times \frac{2}{\text{Number of Distinct Operators}}
\]  
(8)

\[
\text{Halstead Length} = \text{Total Number of Operands} + \text{Total Number of Operators}
\]  
(9)
7.2. Acquired knowledge

Since a decision forest can be represented by a large amount of logic rules, we choose a select few that we find interesting. These rules come from our CSForests built from datasets PC3 and MC2.

If a module's number of blank lines is greater than three and the percentage of the lines in the module that are comments is greater than 10.87% then treating the module as defective is approximately one third the cost of assuming the module is defect free. Out of the 127 records in MC2', 41 fall in this rule. So the rule has a high support which indicates that it is a common occurrence.

If a module's cyclomatic complexity density is less than or equal to 0.5, Halstead difficulty greater than 48 and number of blank lines greater than 9 then treating the module as defective is one fifth the cost of assuming the module is free of defects. Out of the 127 records in MC2.

If a module's Halstead length greater 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.

An interesting piece of knowledge is that if a module has no blank lines whatsoever, then the risk of defects is so small that assuming the module has no defects is just 22% the cost of treating it as defective.

8. Conclusion

It is basically certain that software development projects accumulate defects in the development process. Software defect prediction systems may be used to advise the software developers as to which modules may contain defects. Cost-sensitivity can be incorporated in an effort to make the predictions made optimised for monetary cost to the developers. We present CSForest, a cost-sensitive decision forest algorithm, and a cost-sensitive voting technique. We show that when combined, CSForest and CSVoting produces lower cost predictions than existing techniques. In addition to the proposed cost-sensitive decision forest algorithm and cost-sensitive voting method, we also propose a method for incorporating resampling techniques into CSForest's forest building process. We find indications that doing so through oversampling in datasets with less than 100 defective examples may help combat the under-representation of the defective class in SDP and achieve a lower cost. Our future work involves further investigating the possible interactions and incorporations of resampling techniques into the decision forest building process. The method of incorporating oversampling techniques into CSForest has only been experimented by using SMOTE, Safe-Level-Smote and by adding noise to existing examples. We plan on creating a new oversampling technique which takes advantage of this method of incorporation. This would be done with the aim of further reducing cost.

Code Availability

The Java code for CSForest, CSVoting and BCSForest can be found online at “mikesiers.com/software/” and also at “http://dx.doi.org/http://csusap.csu.edu.au/~zislam/”.

Acknowledgments

The second author of this paper would like to thank the faculty of Business Compact Funding R4 P55 in Charles Sturt University, Australia. We are grateful for the constructive feedback by the reviewers.

References