Information Systems 51 (2015) 62–71

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

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

ARTICLE INFO

Article history:
Received 29 October 2014
Received in revised form
26 February 2015
Accepted 27 February 2015
Recommended by: D. Shasha
Available online 10 March 2015

Keywords:
Software defect prediction
Decision forest
Cost-sensitive
Forest voting
Class imbalance

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.

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

* Corresponding author.
E-mail addresses: mikesiers@live.com (M.J. Siers), zislam@csu.edu.au (M.Z. Islam).

http://dx.doi.org/10.1016/j.is.2015.02.006
0306-4379/© 2015 Elsevier Ltd. All rights reserved.

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

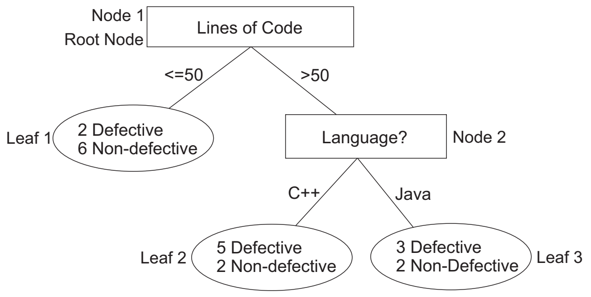

Fig. 1. An example of a decision tree.

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 ≤ 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 six of 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 $\frac{6}{8}$ if the line of code in the module is ≤ 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

Table 1
Cost metric.

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

Table 2
Total classification cost of the existing methods.

Dataset	C4.5	SVM	SysFor +Voting1	SysFor + Voting2	CSC +C4.5	CSTree
MC2′	**154**	162	164	171	**154**	165
PC1′	287	275	**255**	257	283	290
KC1′	1282	1476	1404	1397	1226	**1187**
PC3′	640	623	664	663	637	**586**
MC1′	272	321	275	**264**	272	291
PC2′	85	82	**81**	86	85	84

Table 3
Impact of the proposed CSVoting on SysFor.

Dataset	SysFor +Voting1	SysFor + Voting2	SysFor +CSVoting
MC2′	164	171	**145**
PC1′	**255**	257	271
KC1′	1404	1397	**1268**
PC3′	664	663	**571**
MC1′	275	**264**	270
PC2′	**81**	86	**81**

in one subset there are 2 records having Defective and 6 records having Non-Defective class values, and in the other 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 in both subsets compared to the complete dataset.

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* $\le 50 \Rightarrow$ 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 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_T 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.

Classifying a single unlabeled record R using a forest is not as straightforward as a tree, since a forest typically 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 T_i 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 L^{100} 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 L^{100} 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 L^{100} 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_i^{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]

Table 4
CSForest total cost comparison.

Dataset	C4.5	SVM	SysFor +Voting1	SysFor + Voting2	CSC +C4.5	CSTree	CSForest + CSVoting
MC2′	154	162	164	171	154	165	**129**
PC1′	287	275	**255**	257	283	290	276
KC1′	1282	1476	1404	1397	1226	1187	**1168**
PC3′	640	623	664	663	637	586	**521**
MC1′	272	321	275	264	272	291	**261**
PC2′	85	82	81	86	85	84	**80**

alters the distribution of the class values within a training dataset D_T. For example, if D_T is comprised of 10 records with the *Defective* class value (we call them defective records) and 10 records with the *Non-Defective* class value (we call them non-defective records), Weighting may change the class distribution to 15 defective records and 5 non-defective records. The change of the class distribution is expected to reduce the classification cost since the cost of a False Negative prediction is generally higher than a False Positive prediction. The new class distribution is carefully computed considering the classification cost of the class values, number of records having the class values and total number of records in D_T.

A recent cost-sensitive decision tree algorithm called CSTree [12,16] uses expected classification cost reduction to find the best splitting point. In order to calculate expected classification cost CSTree uses Eqs. (1) and (2). Before discussing the equations we introduce a set of terms and notations as follows.

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 $C_N = N_{TN} \times C_{TN} + N_{FN} \times C_{FN}$:

$$E = \frac{2 \times C_P \times C_N}{C_P + C_N} \tag{1}$$

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^i and C_N^i are the costs of labeling the records within the ith subset as positive and negative, respectively. The expected classification cost for A_i is then computed using the following equation:

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

Finally, the expected classification cost reduction for A_i is given by $E - E_{A_i} - T_C^i$ where T_C^i 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 $E - E_{A_i} - T_C^i > 0$, 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.

Datasets collected for SDP typically suffer from the class imbalance problem, for examples see the NASA MDP datasets [20]. This means that the datasets contain many more non-defective examples than they do defective examples. This poses a problem to data mining algorithms since there exists an under-representation of the defective examples and an over-representation of the non-defective examples. Resampling techniques can be employed by the data miner in order to account for class imbalance. Resampling techniques involve altering the examples in the dataset. This can be achieved by using either undersampling or oversampling techniques.

SMOTE [4] is called a *synthetic* oversampling technique since it creates new synthetic examples rather than duplicating existing ones. For each record in the minority class of a dataset, SMOTE finds its k nearest neighbours. One of those k nearest neighbours N is chosen at random.

Table 5
CSForest total cost comparison with individual folds.

Dataset	C4.5				SVM				SysFor +Voting1				SysFor + Voting2				CSC + C4.5				CSTree				CSForest + CSVoting			
	Fold 1	Fold 2	Fold 3	Total Cost	Fold 1	Fold 2	Fold 3	Total Cost	Fold 1	Fold 2	Fold 3	Total Cost	Fold 1	Fold 2	Fold 3	Total Cost	Fold 1	Fold 2	Fold 3	Total Cost	Fold 1	Fold 2	Fold 3	Total Cost	Fold 1	Fold 2	Fold 3	Total Cost
MC2′	43	50	61	154	41	55	66	162	47	49	68	164	50	49	72	171	43	50	61	154	44	53	68	165	42	32	55	**129**
PC1′	108	87	92	287	90	85	100	275	85	78	92	**255**	86	80	91	257	108	83	92	283	101	84	105	290	99	92	85	276
KC1′	439	426	417	1282	504	493	479	1476	448	445	511	1404	451	440	506	1397	434	383	409	1226	382	370	435	1187	379	356	433	**1168**
PC3′	207	222	211	640	183	220	220	623	243	239	182	664	252	237	184	663	204	222	211	637	215	224	147	586	218	147	156	**521**
MC1′	80	92	100	272	116	107	98	321	71	86	118	275	64	84	116	264	80	92	100	272	77	96	118	291	64	79	118	**261**
PC2′	35	25	25	85	32	29	21	82	10	40	31	81	14	41	31	86	35	25	25	85	10	40	34	84	10	40	30	**80**

For each attribute, the new synthetic record takes a value between the attribute values of the current example and of N. 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 ∞ 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 $L = \{L_1, L_2, \ldots L_n\}$. 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) $E - E_{A_i} - T_C^i$ (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 $E - E_{A_i} - T_C^i > 0$ whereas CSForest modifies the criterion to be $E - E_{A_i} - T_C^i \geq 0$ 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 trees T, goodness threshold τ, separation threshold ϵ, pruning confidence factor c and minimum number of records in a leaf m_l.

each attribute $A_i \in A$. If A_i is numerical then it also finds the best splitting point P_i for A_i. If A_i is categorical then each domain value of A_i is considered to be the splitting point P_i. It then sorts the attributes (along with the corresponding splitting points) in the descending order of their CCR ability where A_1 has the best CCR ability. All attributes that have a CCR ability within a user defined goodness threshold τ of A_1 are then added in the set of good

Algorithm 1. CSForest.

```
Input: $D_T$, $T$, $\tau$, $\epsilon$, $c$, $m_l$
Output: a set of trees $F$
initialise a set of decision trees $F$ to null;
initialise a set of good attributes $A^g$ to null;
initialise a set of split points $P^g$ to null;
set $A^g$, and $P^g$ by calling GetGoodAttributes($D_T$, $\tau$, $\epsilon$);
initialise $i$ to 1;
while $|F| < T$ and $|F| < |A^g|$ do
    $T_i = $ BuildCSTree$(D_T, A_i \in A^g, P_i \in P^g, c, m_l)$;
    $F = F \cup T_i$;
    $i = i+1$;
end
$i = 1$;
initialise $K$ to $|F|$;
while $|F| < T$ and $i \le K$ do
    /* divide the data set $D_T$. */
    if $A_i \in A^g$ is categorical then
        $D_T = \{D_1, D_2, \ldots D_{|A_i|}\}$;
    end
    if $A_i \in A^g$ is numerical then
        $D_T = \{D_1, D_2\}$;
        /*$D_T$ is divided using $A_i, P_i$ */
    end
    /*$\overline{|D_T|}$ = number of data segments in $D_T$, $|D_x|$ = number of records in the xth segment */
    for $j = 1$ to $\overline{|D_T|}$ do
        initialise $A_j^g$, and $P_j^g$ to null;
        $A_j^g$, and $P_j^g$ = GetGoodAttributes$(D_j, \tau, \epsilon)$;
    end
    initialise number of possible trees, $t_p$ to null;
    calculate, $t_p = \frac{\sum_{j=1}^{\overline{|D_T|}} |A_j^g| \times |D_j|}{\sum_{j=1}^{\overline{|D_T|}} |D_j|}$;
    initialise $x$ to1;
    while $|F| < T$ and $x \le t_p$ do
        for $j = 1$ to $\overline{|D_T|}$ do
            if $|A_j^g| > x$ then
                $t_j = $ BuildCSTree$(D_j, a_{x+1} \in A_j^g, p_{x+1} \in P_j^g, c, m_l)$;
            end
            if $|A_j^g| \le x$ then
                $t_j = $ BuildCSTree$(D_j, a_1 \in A_j^g, p_1 \in P_j^g, c, m_l)$;
            end
            Build a tree $T_{new}$ by joining the root node of each $t_j (1 \le j \le \overline{|D_T|})$ as a child node with the root node of $T_i$;
            $F = F \cup T_{new}$;
            $x = x+1$;
        end
    end
    $i = i+1$;
end
return F;
```

Using GetGoodAttributes(D_T, τ, ϵ) CSForest first computes the Classification Cost Reduction (CCR) ability of

attributes A^g. The set P^g stores the corresponding splitting point/s P_i for $A_i \in A^g$. Note that if A_i 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 ϵ (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 $|F| < T$ 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 D_T into the subsets $D_T = \{D_1, D_2, \ldots D_{|A_1|}\}$ based on A_1. For each subset D_i it applies the same approach to find the set of good attributes (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 $A_i \in A^g$ is computed using the number of good attributes in each subset $D_j \in D_T$ 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 $A_2 \in A^g$ and so on until it creates the T trees.

4. Experimental results

Following the common trend in Software Defect 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 classification. 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 available 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.

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

Table 7
Weighted recall comparison.

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

Table 8
Incoporating oversampling into CSForest.

Dataset	CSForest	Setup 1	Setup 2	Setup 3	Setup 4	Setup 5	Setup 6
		CSForest +SMOTE(2)	CSForest + SMOTE(5)	CSForest +SLS(2)	CSForest +SLS(5)	CSForest +Noise(2)	CSForest +Noise(5)
MC2′	129	136	114	137	132	135	122
PC1′	276	250	256	267	271	311	307
KC1′	1168	1206	1200	1212	1206	1228	1215
PC3′	521	555	561	647	607	576	582
MC1′	261	250	264	269	279	247	247
PC2′	80	88	80	87	83	88	88

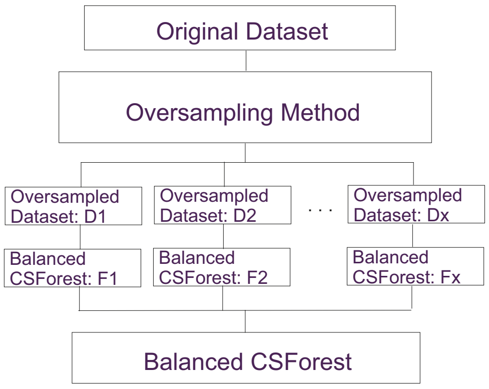

Fig. 2. Incorporating oversampling into CSForest.

Table 9
Datasets class imbalance.

Dataset	Records	Defective/non-defective
MC2′	127	44/83
PC1′	759	61/698
KC1′	2096	325/1771
PC3′	1125	140/985
MC1′	9277	68/9209
PC2′	1585	16/1569

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_{TN}, and C_{FN} 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 $C_{FN}=5$. 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.

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 lower 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 $MC1'$ and $PC2'$ 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 $D1$, $D2$, and Dx. From each of the oversampled data sets, we build a CSForest; $F1$, $F2$, and Fx. 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,

$$\text{Cyclomatic Complexity Density} = \frac{\text{Cyclomatic Complexity}}{\text{Logical Line Count}} \tag{7}$$

$$\text{Halstead Difficulty} = \frac{\text{Number of Distinct Operators}}{2} \times \frac{\text{Total Number of Operands}}{\text{Number of Distinct Operands}} \tag{8}$$

$$\text{Halstead Length} = \text{Total Number of Operands} + \text{Total Number of Operators} \tag{9}$$

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

[1] L. Breiman, Bagging predictors, Mach. Learn. 24 (2) (1996) 123–140.

[2] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32.

[3] C. Bunkhumpornpat, K. Sinapiromsaran, C. Lursinsap, Safe-level-smote: safe-level-synthetic minority over-sampling technique for handling the class imbalanced problem, in: Advances in Knowledge Discovery and Data Mining, 2009, pp. 475–482.

[4] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, SMOTE: synthetic minority over-sampling technique, J. Artif. Intell. Res. 16 (2002) 321–357.

[5] P. Domingos, Metacost: a general method for making classifiers cost-sensitive, in: Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, San Diego, CA, USA, 1999, pp. 155–164.

[6] Y. Freund, R.E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, J. Comput. Syst. Sci. 55 (1) (1997) 119–139.

[7] D. Gray, D. Bowes, N. Davey, Y. Sun, B. Christianson, The misuse of the nasa metrics data program data sets for automated software defect prediction, in: 15th Annual Conference on Evaluation and Assessment in Software Engineering (EASE 2011), IET, Durham, UK, 2011, pp. 96–103.

[8] D. Gray, D. Bowes, N. Davey, Y. Sun, B. Christianson, Reflections on the nasa mdp data sets, IET Softw. 6 (6) (2012) 549–558.

[9] M.H. Halstead, Elements of Software Science, North-Holland, New York, 1977.

[10] G. Holmes, A. Donkin, I.H. Witten, Weka: A machine learning workbench, in: Proceedings of the Second Australian and New Zealand Conference on Intelligent Information Systems, IEEE, Brisbane, Australia, 1994, pp. 357–361.

[11] M.Z. Islam, H. Giggins, Knowledge discovery through sysfor: a systematically developed forest of multiple decision trees, in: Proceedings of the Ninth Australasian Data Mining Conference, vol. 121, Australian Computer Society, Inc., Ballarat, Australia, 2011, pp. 195–204.

[12] C.X. Ling, V.S. Sheng, T. Bruckhaus, N.H. Madhavji, Maximum profit mining and its application in software development, in: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, Philadelphia, USA, 2006, pp. 929–934.

[13] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. 4 (1976) 308–320.

[14] G.K. Gill, C.F. Kemerer, Cyclomatic complexity density and software maintenance productivity, IEEE Trans. Softw. Eng. 17 (12) (1991) 1284–1288.

[15] J.R. Quinlan, C4. 5: Programs for Machine Learning, vol. 1, Morgan kaufmann, San Franciso, USA, 1993.

[16] V.S. Sheng, B. Gu, W. Fang, J. Wu, Cost-sensitive learning for defect escalation, Knowl.-Based Syst. 66 (2014) 146–155.

[17] J. Platt, Fast Training of Support Vector Machines using Sequential Minimal Optimization, in: B. Scholkopf, C.J.C. Burges, A.J. Smola (Eds.), Advances in Kernel Methods—Support Vector Learning, MIT Press, Cambridge, MA., 1999, pp. 185–208.

[18] S.S. Keerthi, S.K. Shevade, C. Bhattacharyya, K.R.K. Murthy, Improvements to Platt's SMO algorithm for SVM classifier design, Neural Comput. 13 (3) (2001).

[19] V.S. Sheng, C.X. Ling, Thresholding for making classifiers cost-sensitive, in: Proceedings of the National Conference on Artificial Intelligence, vol. 21, AAAI Press, MIT Press, Menlo Park, CA, Cambridge, MA, London, 2006, pp. 476–481.

[20] M. Shepperd, Nasa-Software Defect Datasets, Online (accessed 15.07.14).

[21] M. Shepperd, Q. Song, Z. Sun, C. Mair, Data quality: some comments on the nasa software defect datasets, IEEE Trans. Softw. Eng. 39 (9) (2013) 1208–1215.

[22] M.J. Siers, M.Z. Islam, Cost sensitive decision forest and voting for software defect prediction, in: Proceedings of the PRICAI 2014: Trends in Artificial Intelligence, Springer International Publishing, Gold Coast, Australia, 2014, pp. 929–936.

[23] K.M. Ting, An instance-weighting method to induce cost-sensitive trees, IEEE Trans. Knowl. Data Eng. 14 (3) (2002) 659–665.