Data Mining and Knowledge Discovery, 4, 315–344, 2000

PUBLIC: A Decision Tree Classifier that Integrates Building and Pruning

RAJEEV RASTOGI rastogi@bell-labs.com
Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974, USA

KYUSEOK SHIM shim@cs.kaist.ac.kr
Korea Advanced Institute of Science and Technology, and Advanced Information Technology Research Center, 373-1 Kusong-dong, Yusong-gu, Taejon 305-701, South Korea

Editors: Fayyad, Mannila, Ramakrishnan

Abstract. Classification is an important problem in data mining. Given a database of records, each with a class label, a classifier generates a concise and meaningful description for each class that can be used to classify subsequent records. A number of popular classifiers construct decision trees to generate class models. These classifiers first build a decision tree and then prune subtrees from the decision tree in a subsequent *pruning* phase to improve accuracy and prevent "overfitting".

Generating the decision tree in two distinct phases could result in a substantial amount of wasted effort since an entire subtree constructed in the first phase may later be pruned in the next phase. In this paper, we propose PUBLIC, an improved decision tree classifier that integrates the second "pruning" phase with the initial "building" phase. In PUBLIC, a node is not expanded during the building phase, if it is determined that it will be pruned during the subsequent pruning phase. In order to make this determination for a node, before it is expanded, PUBLIC computes a lower bound on the minimum cost subtree rooted at the node. This estimate is then used by PUBLIC to identify the nodes that are certain to be pruned, and for such nodes, not expend effort on splitting them. Experimental results with real-life as well as synthetic data sets demonstrate the effectiveness of PUBLIC's integrated approach which has the ability to deliver substantial performance improvements.

Keywords: data mining, classification, decision tree

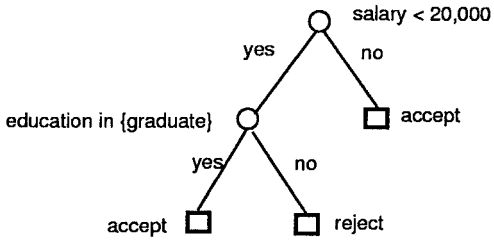
1. Introduction

Classification is an important problem in data mining. It has been studied extensively by the machine learning community as a possible solution to the *knowledge acquisition* or *knowledge extraction* problem. The input to the classifier construction algorithm is a *training set* of records, each of which is tagged with a class label. A set of attribute values defines each record. Attributes with discrete domains are referred to as *categorical*, while those with ordered domains are referred to as *numeric*. The goal is to induce a model or description for each class in terms of the attributes. The model is then used by the classifier to classify future records whose classes are unknown.

Figure 1(a) shows an example training set for a loan approval application. There is a single record corresponding to each loan request, each of which is tagged with one of two labels—accept if the loan request is approved or reject if the loan request is denied. Each

salary	*education*	label
10,000	high-school	reject
40,000	under-graduate	accept
15,000	under-graduate	reject
75,000	graduate	accept
18,000	graduate	accept

(a)



(b)

Figure 1. Decision trees.

record is characterized by two attributes, *salary* and *education*, the former numeric and the latter categorical with domain {high-school, undergraduate, graduate}. The attributes denote the income and the education level of the loan applicant. The goal of the classifier is to deduce, from the training data, concise and meaningful conditions involving *salary* and *education* under which a loan request is accepted or rejected.

Classification has been successfully applied to several areas like medical diagnosis, weather prediction, credit approval, customer segmentation and fraud detection. Among the techniques developed for classification, popular ones include bayesian classification (Cheeseman et al., 1988), neural networks (Bishop, 1995; Ripley, 1996), genetic algorithms (Goldberg, 1989) and decision trees (Breiman et al., 1984): In this paper, however, we focus on decision trees. There are several reasons for this. First, compared to a neural network or a bayesian classification based approach, a decision tree is easily interpreted/comprehended by humans (Breiman et al., 1984). Second, while training neural networks can take large amounts of time and thousands of iterations, inducing decision trees is efficient and is thus suitable for large training sets. Also, decision tree generation algorithms do not require additional information besides that already contained in the training data (e.g., domain knowledge or prior knowledge of distributions on the data or classes) (Fayyad, 1991). Finally, as shown in Mitchie et al. (1994), decision trees display good classification accuracy compared to other techniques.

Figure 1(b) is a decision tree for the training data in figure 1(a). Each internal node of the decision tree has a test involving an attribute, and an outgoing branch for each possible outcome. Each leaf has an associated class. In order to classify new records using a decision tree, beginning with the root node, successive internal nodes are visited until a leaf is reached. At each internal node, the test for the node is applied to the record. The outcome of the test at an internal node determines the branch traversed, and the next node visited. The class for the record is simply the class of the final leaf node. Thus, the conjunction of all the conditions for the branches from the root to a leaf constitute one of the conditions for the class associated with the leaf. For instance, the decision tree in figure 1(b) approves a loan request only if *salary* $\geq$ 20,000 or *education* $\in$ {graduate}; otherwise, it rejects the loan application.

A number of algorithms for inducing decision trees have been proposed over the years (e.g., CLS (Hunt et al., 1966), ID3 (Quinlan, 1986), C4.5 (Quinlan, 1993), CART (Breiman et al., 1984), SLIQ (Mehta et al., 1996), SPRINT (Shafer et al., 1996)). Most of the algorithms have two distinct phases, a *building* or *growing* phase followed by a *pruning* phase. In the building phase, the training data set is recursively partitioned until all the records in a partition have the same class. For every partition, a new node is added to the decision tree; initially, the tree has a single root node for the entire data set. For a set of records in a partition P, a test criterion T for further partitioning the set into $P_1, \ldots, P_m$ is first determined. New nodes for $P_1, \ldots, P_m$ are created and these are added to the decision tree as children of the node for P. Also, the node for P is labeled with test T, and partitions $P_1, \ldots, P_m$ are then recursively partitioned. A partition in which all the records have identical class labels is not partitioned further, and the leaf corresponding to it is labeled with the class.

The building phase constructs a perfect tree that accurately classifies every record from the training set. However, one often achieves greater accuracy in the classification of new objects by using an imperfect, smaller decision tree rather than one which perfectly classifies all known records (Quinlan and Rivest, 1989). The reason is that a decision tree which is perfect for the known records may be overly sensitive to statistical irregularities and idiosyncrasies of the training set. Thus, most algorithms perform a pruning phase after the building phase in which nodes are iteratively pruned to prevent "overfitting" and to obtain a tree with higher accuracy.

An important class of pruning algorithms are those based on the Minimum Description Length (MDL) principle (Quinlan and Rivest, 1989; Wallace and Patrick, 1993; Fayyad and Irani, 1993; Mehta et al., 1995). Consider the problem of communicating the classes for a set of records. Since a decision tree partitions the records with a goal of separating those with similar class labels, it can serve as an efficient means for encoding the classes of records. Thus, the "best" decision tree can then be considered to be the one that can communicate the classes of the records with the "fewest" number of bits. The cost (in bits) of communicating classes using a decision tree comprises of (1) the bits to encode the structure of the tree itself, and (2) the number of bits needed to encode the classes of records in each leaf of the tree. We thus need to find the tree for which the above cost is minimized. This can be achieved as follows. A subtree S is pruned if the cost of directly encoding the records in S is no more than the cost of encoding the subtree plus the cost of the records in each leaf of the subtree. In Mehta et al. (1995), it is shown that MDL pruning (1) leads to accurate trees for a wide range of data sets, (2) produces trees that are significantly smaller in size, and (3) is computationally efficient and does not use a separate data set for pruning. For the above reasons, the pruning algorithms developed in this paper employ MDL pruning.

Generating the decision tree in two distinct phases could result in a substantial amount of wasted effort since an entire subtree constructed in the first phase may later be pruned in the next phase. During the building phase, before splitting a node, if it can be concluded that the node will be pruned from the tree during the subsequent pruning phase, then we could avoid building the subtree rooted at the node. Consequently, since building a subtree usually requires repeated scans to be performed over the data, significant reductions in I/O and improvements in performance can be realized. In this paper, we present PUBLIC

(PrUning and BuiLding Integrated in Classification), a decision tree classifier that during the growing phase, first determines if a node will be pruned during the following pruning phase, and subsequently stops expanding such nodes. Thus, PUBLIC integrates the pruning phase into the building phase instead of performing them one after the other. Furthermore, by only pruning nodes that we know will definitely be pruned in the pruning phase, we guarantee that the tree generated by PUBLIC's integrated approach is exactly the same as the tree that would be generated as a result of executing the two phases separately, one after another.

Determining, during the building phase, whether a node will be pruned during the pruning phase is problematic since the tree is only partially generated. Specifically, this requires us to estimate at each leaf of the partial tree, based on the records contained in the leaf, a lower bound on the cost of the subtree rooted at the leaf. Furthermore, the better (higher) this estimate, the more we can prune during the building phase and consequently, the more we can improve performance. We present several algorithms for estimating the subtree cost—the algorithms illustrate the trade-off between accuracy of the estimate and the computation involved. Our experimental results on real-life as well as synthetic data sets demonstrate that PUBLIC's integrated approach can result in substantial performance improvements compared to traditional classifiers.

The remainder of the paper is organized as follows. In Section 2, we survey existing work on decision tree classifiers. Details of the building and pruning phases of a traditional decision tree classifier along the lines of SPRINT (Shafer et al., 1996) are presented in Section 3. The PUBLIC algorithm as well as techniques for estimating lower bounds on subtree costs are described in Sections 4 and 5. In Section 6, we compare PUBLIC's performance with that of a traditional decision tree classifier. Finally, in Section 7, we offer concluding remarks.

2. Related work

In this section, we provide a brief survey of related work on decision tree classifiers. The growing phase for the various decision tree generation systems differ in the algorithm employed for selecting the test criterion T for partitioning a set of records. CLS (Hunt et al., 1966), one of the earliest systems, examines the solution space of all possible decision trees to some fixed depth. It then chooses a test that minimizes the cost of classifying a record. The cost is made up of the cost of determining the feature values for testing as well as the cost of misclassification. ID3 (Quinlan, 1986) and C4.5 (Quinlan, 1993) replace the computationally expensive look-ahead scheme of CLS with a simple information theory driven scheme that selects a test that minimizes the *information entropy* of the partitions (we discuss entropy further in Section 3), while CART (Breiman et al., 1984), SLIQ (Mehta et al., 1996) and SPRINT (Shafer et al., 1996) select the test with the lowest GINI index. Classifiers like C4.5 and CART assume that the training data fits in memory. SLIQ and SPRINT, however, can handle large training sets with several million records. SLIQ and SPRINT achieve this by maintaining separate lists for each attribute and pre-sorting the lists for numeric attributes. We present a detailed description of SPRINT, a state of the art classifier for large databases, in Section 3.

In addition to MDL pruning described earlier, there are two other broad classes of pruning algorithms. The first includes algorithms like cost-complexity pruning (Quinlan, 1987) that first generate a sequence of trees obtained by successively pruning non-leaf subtrees for whom the ratio of the reduction in misclassified objects due to the subtree and the number of leaves in the subtree is minimum. A second phase is then carried out in which separate pruning data (distinct from the training data used to grow the tree) is used to select the tree with the minimum error. In the absence of separate pruning data, cross-validation can be used at the expense of a substantial increase in computation. The second class of pruning algorithms, pessimistic pruning (Quinlan, 1987), do not require separate pruning data, and are computationally inexpensive. Experiments have shown that this pruning leads to trees that are "too" large with high error rates.

The above-mentioned decision tree classifiers only consider "guillotine-cut" type tests for numeric attributes. Since these may result in very large decision trees when attributes are correlated, in Fukuda et al. (1996), the authors propose schemes that employ tests involving two (instead of one) numeric attributes and consider partitions corresponding to grid regions in the two-dimensional space. Recently, in Gehrke et al. (1998), the authors propose Rainforest, a framework for developing fast and scalable algorithms for constructing decision trees that gracefully adapt to the amount of main memory available. In Fayyad and Irani (1993), Zihed et al. (1997), the authors use the entropy minimization heuristic and MDL principle for discretizing the range of a continuous-valued attribute into multiple intervals.

Note that PUBLIC's integrated approach is different from that in Agrawal et al. (1992) where a *dynamic pruning* criterion based on pessimistic pruning is used to stop expanding nodes during the growing phase. The dynamic pruning scheme proposed in Agrawal et al. (1992) is ad-hoc and it does not guarantee that the resulting tree is the same as the tree that would be obtained as a result of performing the pruning phase after the completion of the building phase. This is a major drawback and could adversely impact the accuracy of the tree. For instance, in Agrawal et al. (1992), if successive expansions of a node and its children do not result in acceptable error reduction, then further expansions of its children are terminated.

3. Preliminaries

In this section, we present a more detailed description of the building and pruning phases of a traditional decision tree classifier. In the following subsections, the tree building phase is identical to that used in SPRINT (Shafer et al., 1996), while the MDL pruning algorithm employed for pruning the tree is along the lines described in Quinlan and Rivest (1989) and Mehta et al. (1995). While traditional classifiers perform pruning only after the tree has been completely built, in PUBLIC, the building and pruning phases are interleaved.

3.1. Tree building phase

The overall algorithm for building a decision tree is as shown in figure 2. The tree is built breadth-first by recursively partitioning the data until each partition is *pure*, that is, each

```
procedure buildTree(S):
1.  Initialize root node using data set S
2.  Initialize queue Q to contain root node
3.  while Q is not empty do {
4.     dequeue the first node N in Q
5.     if N is not pure {
6.        for each attribute A
7.           Evaluate splits on attribute A
8.        Use best split to split node N into N1 and N2
9.        Append N1 and N2 to Q
10.    }
11. }
```

Figure 2. Building algorithm.

partition contains records belonging to the same class. The splitting condition for partitioning the data is either of the form $A < v$ if A is a numeric attribute (v is a value in the domain of A) or $A \in V$ if A is a categorical attribute (V is a set of values from A's domain). Thus, each split is binary.

Data structures. Each node of the decision tree maintains a separate list for every attribute. Each attribute list contains a single entry for every record in the partition for the node. The attribute list entry for a record contains three fields—the value for the attribute in the record, the class label for the record and the record identifier. Attribute lists for the root node are constructed at the start using the input data, while for other nodes, they are derived from their parent's attribute lists when the parent nodes are split. Attribute lists for numeric attributes at the root node are sorted initially and this sort order is preserved for other nodes by the splitting procedure. Also, at each node, a histogram is maintained that captures the class distribution of the records at the node. Thus, the initialization of the root node in Step 1 of the build algorithm involves (1) constructing the attribute lists, (2) sorting the attribute lists for numeric attributes, and (3) constructing the histogram for the class distribution.

Selecting splitting attribute and split point. For a set of records S, the entropy $E(S)$ is defined as—$\sum_j p_j \log p_j$, where p_j is the relative frequency of class j in S. Thus, the more homogeneous a set is with respect to the classes of records in the set, the lower is its entropy. The entropy of a split that divides S with n records into sets S_1 with n_1 records and S_2 with n_2 records is $E(S_1, S_2) = \frac{n_1}{n}E(S_1) + \frac{n_2}{n}E(S_2)$. Consequently, the split with the least entropy best separates classes, and is thus chosen as the best split for a node.

To compute the best split point for a numeric attribute, the (sorted) attribute list is scanned from the beginning and for each split point, the class distribution in the two partitions is determined using the class histogram for the node. The entropy for each split point can thus be efficiently computed since the lists are stored in a sorted order. For categorical attributes, the attribute list is scanned to first construct a histogram containing the class distribution

for each value of the attribute. This histogram is then utilized to compute the entropy for each split point.

We must point out that instead of entropy, other criteria can be used to select the best split point for an attribute. For instance, SPRINT uses the *gini index* which, for a set of records S, is defined as $1 - \sum_j p_j^2$, where p_j is the relative frequency of class j in S. Additional information on the gini index can be found in Shafer et al. (1996).

Splitting attribute lists. Once the best split for a node has been found, it is used to split the attribute list for the splitting attribute amongst the two child nodes. Each record identifier along with information about the child node that it is assigned to (left or right) is then inserted into a hash table. The remaining attribute lists are then split using the record identifier stored with each attribute list entry and the information in the hash table. Class distribution histograms for the two child nodes are also computed during this step.

3.2. Tree pruning phase

To prevent overfitting, the MDL principle (Rissanen, 1978; Rissanen, 1989) is applied to prune the tree built in the growing phase and make it more general. The MDL principle states that the "best" tree is the one that can be encoded using the fewest number of bits. Thus, the challenge for the pruning phase is to find the subtree of the tree that can be encoded with the least number of bits.

In the following, we first present a scheme for encoding decision trees. We then present a pruning algorithm that, in the context of our encoding scheme, finds the minimum cost subtree of the tree constructed in the growing phase. In the remainder of the paper, we assume that a is the number of attributes.

Cost of encoding data records. Let a set S contain n records each belonging to one of k classes, n_i being the number of records with class i. The cost of encoding the classes for the n records (Quinlan and Rivest, 1989) is given by[1]

$$\log \binom{n+k-1}{k-1} + \log \frac{n!}{n_1! \cdots n_k!}$$

In the above equation, the first term is the number of bits to specify the class distribution, that is, the number of records with classes $1, \ldots, k$. The second term is the number of bits required to encode the class for each record once it is known that there are n_i records with class label i. In Mehta et al. (1995), it is pointed out that the above equation is not very accurate when some of the n_i are either close to zero or close to n. Instead, they suggest using the following equation from Krichevsky and Trofimov (1981), which is what we adopt in this paper for the cost $C(S)$ of encoding the classes for the records in set S.

$$C(S) = \sum_i n_i \log \frac{n}{n_i} + \frac{k-1}{2} \log \frac{n}{2} + \log \frac{\pi^{k/2}}{\Gamma(k/2)} \tag{1}$$

In Eq. (1), the first term is simply $n * E(S)$, where $E(S)$ is the entropy of the set S of records. Also, since $k \leq n$, the sum of the last two terms in Eq. (1) is always non-negative. We utilize this property later in the paper when computing a lower bound on the cost of encoding the records in a leaf.

In SPRINT, the cost of encoding a set of data records is assumed to be simply the number of records that do not belong to the majority class for the set. However, our experience with most real-life data sets has been that using Eq. (1) instead results in more accurate trees. In PUBLIC, even though we use Eq. (1) as the cost for encoding a set of records, PUBLIC's pruning techniques are also applicable if we were to use the approach adopted in SPRINT.

Cost of encoding tree. The cost of encoding the tree comprises of three separate costs:

1. The cost of encoding the structure of the tree.
2. The cost of encoding for each split, the attribute and the value for the split.
3. The cost of encoding the classes of data records in each leaf of the tree.

The structure of the tree can be encoded by using a single bit in order to specify whether a node of the tree is an internal node (1) or leaf (0). Thus, the bit string 11000 encodes the tree in figure 1(b). Since we are considering only binary decision trees, the proposed encoding technique for representing trees is nearly optimal (Quinlan and Rivest, 1989).

The cost of encoding each split involves specifying the attribute that is used to split the node and the value for the attribute. The splitting attribute can be encoded using $\log a$ bits (since there are a attributes), while specifying the value depends on whether the attribute is categorical or numeric. Let v be the number of distinct values for the splitting attribute in records at the node. If the splitting attribute is numeric, then since there are $v - 1$ different points at which the node can be split, $\log(v - 1)$ bits are needed to encode the split point. On the other hand, for a categorical attribute, there are 2^v different subsets of values of which the empty set and the set containing all the values are not candidates for splitting. Thus, the cost of the split is $\log(2^v - 2)$. For an internal node N, we denote the cost of describing the split by $C_{split}(N)$.

Finally, the cost of encoding the data records in each leaf is as described in Eq. (1).

Pruning algorithm. Now that we have a formulation for the cost of a tree, we next turn our attention to computing the minimum cost subtree of the tree constructed in the building phase. The simple recursive algorithm in figure 3 computes the minimum cost subtree rooted at an arbitrary node N and returns its cost. Let S be the set of records associated with N. If N is a leaf, then the minimum cost subtree rooted at N is simply N itself. Furthermore, the cost of the cheapest subtree rooted at N is $C(S) + 1$ (we require 1 bit in order to specify that the node is a leaf).

On the other hand, if N is an internal node in the tree with children N_1 and N_2, then there are the following two choices for the minimum cost subtree—(1) the node N itself with no children (this corresponds to pruning its two children from the tree, thus making node N a leaf), or (2) node N along with children N_1 and N_2 and the minimum cost subtrees rooted at N_1 and N_2. Of the two choices, the one with the lower cost results in the minimum cost subtree for N.

```
procedure computeCost&Prune(Node N):
    /* S is the set of data records for N */
1.  if N is a leaf return (C(S) + 1)
    /* N1 and N2 are N's children */
2.  minCost1 := computeCost&Prune(N1);
3.  minCost2 := computeCost&Prune(N2);
4.  minCostN := min{C(S) + 1, Csplit(N) + 1 + minCost1 + minCost2};
5.  if minCostN = C(S) + 1
6.     prune child nodes N1 and N2 from tree
7.  return minCostN
```

Figure 3. Pruning algorithm.

The cost for choice (1) is $C(S)+1$. In order to compute the cost for choice (2), in Steps 2 and 3, the procedure recursively invokes itself in order to compute the minimum cost subtrees for its two children. The cost for choice (2) is then $C_{split}(N) + 1 + \text{minCost}_1 + \text{minCost}_2$. Thus, the cost of the cheapest subtree rooted at N is given by minCost_N as computed in Step 4. Note that if choice (1) has a smaller cost, then the children of node N must be pruned from the tree. Or stated alternately, children of a node N are pruned if the cost of directly encoding the data records at N does not exceed the cost of encoding the minimum cost subtree rooted at N.

The tree built in the "growing" phase is pruned by invoking the pruning algorithm in figure 3 on the root node.

4. The PUBLIC integrated algorithm

Most algorithms for inducing decision trees perform the pruning phase only after the entire tree has been generated in the initial building phase. Our typical experience on real-life data sets has been that the pruning phase prunes large portions of the original tree—in some cases, this can be as high as 90% of the nodes in the tree (see Section 6). These smaller trees are more general and result in smaller classification error for records whose classes are unknown (Quinlan and Rivest, 1989; Fayyad, 1991).

It is clear that in most decision tree algorithms, a substantial effort is "wasted" in the building phase on growing portions of the tree that are subsequently pruned in the pruning phase. Consequently, if during the building phase, it were possible to "know" that a certain node is definitely going to be pruned, then we can stop expanding the node further, and thus avoid the computational and I/O overhead involved in processing the node. As a result, by incorporating the pruning "knowledge" into the building phase, it is possible to realize significant improvements in performance. This is the approach followed by the PUBLIC classification algorithm that combines the pruning phase with the building phase.

The PUBLIC algorithm is similar to the build procedure shown in figure 2. The only difference is that periodically or after a certain number of nodes are split (this is a user-defined

parameter), the partially built tree is pruned. The pruning algorithm in figure 3, however, cannot be used to prune the partial tree.

The problem with applying the pruning procedure at a leaf in figure 3 before the tree has been completed is that in the procedure, the cost of the cheapest subtree rooted at a leaf N assumed to be $C(S)+1$. While this is true for a tree that has been completely built, it is not true for a partially built tree since a leaf in a partial tree may be split later, thus becoming an internal node. Consequently, the cost of the subtree rooted at N could be a lot less than $C(S)+1$ as a result of the splitting. Thus, $C(S)+1$ may over-estimate the cost of the cheapest subtree rooted at N and this could resulting in over-pruning, that is, nodes may be pruned during the building phase that would not have been pruned during the pruning phase. This is undesirable since we would like the decision tree induced by PUBLIC to be identical to the one constructed by a traditional classifier.

In order to remedy the above problem, we make use of the following observation—running the pruning algorithm described in figure 3, while under-estimating the minimum cost of subtrees rooted at leaf nodes that can still be expanded, is not harmful. With an under-estimate of the minimum subtree cost at nodes, the nodes pruned are a subset of those that would have been pruned anyway during the pruning phase. PUBLIC's pruning algorithm, illustrated in figure 4, is based on this under-estimation strategy for the cost of the cheapest subtree rooted at a "yet to be expanded" leaf node.

PUBLIC's pruning distinguishes among three kinds of leaf nodes. The first kind of leaves are those that still need to be expanded. For such leaves, as described in the following section, PUBLIC computes a lower bound on the cost of subtrees at the leaves. The two other kinds of leaf nodes consist of those that are either a result of pruning or those that cannot be expanded any further (because they are pure). For such leaves, we use the usual cost of $C(S)+1$. Thus, the pruning procedure is similar to the earlier procedure in figure 3 except that the cost for the cheapest subtree at leaf nodes that have not yet been expanded may not be $C(S)+1$. Also, when children of a node N is pruned, all its descendants are

```
procedure computeCost&PrunePublic(Node N):
    /* S is the set of data records for N */
1.  if N is a "yet to be expanded" leaf return lower bound on subtree cost at N
2.  if N is a "pruned" or "not expandable" leaf return (C(S) + 1)
    /* N1 and N2 are N's children */
3.  minCost1 := computeCost&PrunePublic(N1);
4.  minCost2 := computeCost&PrunePublic(N2);
5.  minCostN := min{C(S) + 1, Csplit(N) + 1 + minCost1 + minCost2};
6.  if minCostN = C(S) + 1 {
7.     Prune child nodes N1 and N2 from tree
8.     Delete nodes N1 and N2 and all their descendants from Q
9.     Mark node N as pruned
10. }
11. return minCostN
```

Figure 4. PUBLIC's pruning algorithm.

removed from the queue Q maintained in the build procedure—this ensures that they are not expanded by the build procedure.

In PUBLIC, the modified pruning algorithm shown in figure 4 is invoked from the build procedure periodically on the root of the partially built tree. Note that once the building phase ends, there are no leaf nodes belonging to the "yet to be expanded" category. As a result, applying the pruning algorithm in figure 4 at the end of the building phase has the same effect as applying the original pruning algorithm and results in the same pruned tree as would have resulted due to the previous pruning algorithm.

5. Computation of lower bound on subtree cost

Any subtree rooted at a node N must have a cost of at least 1, and thus 1 is a simple, but conservative estimate for the cost of the cheapest subtree at leaf nodes that are "yet to be expanded". In our experiments, we found that even with this simple estimate enables PUBLIC to substantially reduce the number of nodes generated. Since more accurate estimates can enable PUBLIC to prune even more nodes, in this section, we propose two algorithms for computing better and higher estimates for the minimum cost subtrees at leaf nodes. The first considers split costs and the second incorporates even the cost of describing the value for each split in the computed estimates. An important point to note is that the computed estimates represent more accurate *lower bounds* on the cost of the cheapest subtree at a leaf node. As mentioned before, it is essential that we underestimate the costs at leaf nodes to prevent over-pruning.

We refer to the version of the PUBLIC algorithm based on a cost estimate of 1 as PUBLIC(1). The other two versions (presented in the following two subsections) that incorporate the cost of splits and cost of values into the estimates are referred to as PUBLIC(S) and PUBLIC(V), respectively. Note that PUBLIC(1), PUBLIC(S) and PUBLIC(V) are identical except for the value returned in Step. 1 of the pruning algorithm in figure 4. Thus, PUBLIC(1), PUBLIC(S) and PUBLIC(V) use increasingly accurate cost estimates for "yet to be expanded" leaf nodes, and result in fewer nodes being expanded during the building phase. We describe the notation employed in the following subsections in Table 1.

Table 1. Notation.

Symbol	Description
S	Set of records in node N
k	Number of classes for the records in S
a	Number of attributes
S_{ij}	Set of records belonging to class i in leaf j
n_{ij}	Number of records belonging to class i in leaf j, that is $\|S_{ij}\|$
S_i	Set of records belonging to class i
n_i	Number of records belonging to class i, that is $\|S_i\|$
S_j	Set of records in leaf j
c_j	Majority class for leaf j

5.1. *Estimating split costs*

The PUBLIC(S) algorithm takes into account split costs when computing a lower bound on the cost of the cheapest subtree rooted at a "yet to be expanded" leaf node N. Specifically, for values of $s \geq 0$, it computes a lower bound on the cost of subtrees rooted at N and containing s splits (and consequently, s internal nodes). The cost estimate for the cheapest subtree at node N is then set to the minimum of the lower bounds computed for the different s values—this guarantees that PUBLIC(S) underestimates the cost of the cheapest subtree rooted at N.

Let S be the set of records at node N and k be the number of classes for the records in S. Also, let n_i be the number of records belonging to class i in S, and $n_i \geq n_{i+1}$ for $1 \leq i < k$ (that is, $n_1, \ldots, n_k$ are sorted in the decreasing order of their values). As before, a denotes the number of attributes. In case node N is not split, that is, $s = 0$, then the minimum cost for a subtree at N is $C(S) + 1$. For values of $s > 0$, a lower bound on the cost of encoding a subtree with s splits and rooted at node N is derived in the following theorem.

Theorem 5.1. *The cost of any subtree with s splits and rooted at node N is at least* $2 * s + 1 + s * \log a + \sum_{i=s+2}^{k} n_i$.

Proof: The cost of encoding the structure of a subtree with s splits is $2 * s + 1$ since a subtree with s splits has s internal nodes and $s + 1$ leaves, and we require one bit to specify the type for each node. Each split also has a cost of at least $\log a$ to specify the splitting attribute. The final term is the cost of encoding the data records in the $s + 1$ leaves of the subtree.

Let n_{ij} denote the number of records belonging to class i in leaf j of the subtree. A class i is referred to as a *majority* class in leaf j if $n_{ij} \geq n_{kj}$ for every other class k in leaf j (in case for two classes i and k, $n_{ij} = n_{kj}$, then one of them is arbitrarily chosen as the majority class). Thus, each leaf has a single majority class, and every other class in the leaf that is not a majority class is referred to as a *minority* class. Since there are $s + 1$ leaves, there can be at most $s + 1$ *majority* classes, and at least $k - s - 1$ classes are a minority class in every leaf.

Consider a class i that is a minority class in leaf j. Due to Eq. (1), $C(S_j)$ the cost of encoding the classes of records in the leaf is at least $\sum_i n_{ij} * E(S_j)$ where S_j is the set of records in leaf j and $\sum_i n_{ij}$ is the total number of records in leaf j. Since for class i, $E(S_j)$ contains the term $\frac{n_{ij}}{\sum_i n_{ij}} \log \frac{\sum_i n_{ij}}{n_{ij}}$, the records of class i in leaf j contribute at least $(\sum_i n_{ij}) * (\frac{n_{ij}}{\sum_i n_{ij}} \log \frac{\sum_i n_{ij}}{n_{ij}})$ to $C(S_j)$. Furthermore, since class i is a minority in leaf j, we have $\frac{\sum_i n_{ij}}{n_{ij}} \geq 2$ and so the records with class i in leaf j contribute at least n_{ij} to $C(S_j)$. Thus, if L is the set containing the $k - s - 1$ classes that are a minority in every leaf, then the minority classes i in L across all the leaves contribute $\sum_{i \in L} n_i$ to the cost of encoding the data records in the leaves of the subtree.

Since we are interested in a lower bound on the cost of the subtree, we need to consider the set L containing $k - s - 1$ classes for which $\sum_{i \in L} n_i$ is minimum. Obviously, the above cost is minimum for the $k - s - 1$ classes with the smallest number of records in S, that

is, classes $s+2, \ldots, k$. Thus, the cost for encoding the records in the $s+1$ leaves of the subtree is at least $\sum_{i=s+2}^{k} n_i$. □

Example 5.2. Consider a database with two attributes *age* and *car type*. Attribute *age* is a numeric attribute, while *car type* is categorical with domain {family, truck, sports}. Also, each record has a class label that is one of low, medium, or high, and which indicates the risk level for the driver. Let a "yet to be expanded" leaf node N contain the following set S of data records.

age	*car type*	label
16	truck	high
24	sports	high
32	sports	medium
34	truck	low
65	family	low

The minimum cost subtrees at N with 1 and 2 splits are as shown in figure 5(a) and figure 5(b), respectively. The minimum cost for encoding each node is presented next to it and the records in each leaf node are listed. Each node has a cost of 1 for encoding its type. In addition, internal nodes have an additional cost of log 2 for specifying the splitting attribute. Furthermore, in figure 5(a), the second leaf node contains a record with class medium which is different from the class for the leaf, and it thus has an extra cost of at least 1. The remaining leaf nodes in both subtrees are all pure nodes and so do not incur any additional costs.

The minimum cost of each subtree is the sum of the minimum costs for all the nodes. Thus, a lower bound on subtrees with 1 split is 5, while for subtrees with 2 splits, it is 7, which are identical to the lower bounds for the subtree costs due to Theorem 5.1.

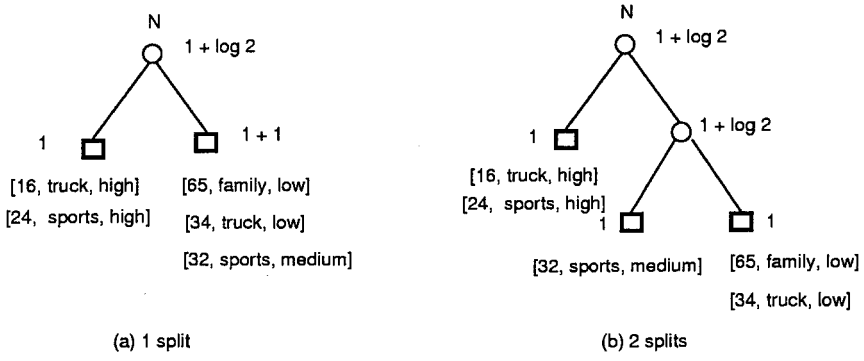


Figure 5. Minimum cost subtrees with 1 and 2 splits.

```
procedure computeMinCostS(Node N):
    /* n_1, ..., n_k are sorted in decreasing order */
1.  if k = 1 return (C(S) + 1)
2.  s := 1
3.  tmpCost := 2 * s + 1 + s * log a + sum_{i=s+2}^{k} n_i
4.  while s + 1 < k and n_{s+2} > 2 + log a do {
5.     tmpCost := tmpCost + 2 + log a - n_{s+2}
6.     s++
7.  }
8.  return min{C(S) + 1, tmpCost}
```

Figure 6. Algorithm for computing lower bound on subtree cost.

Theorem 5.1 gives a lower bound on the cost of any subtree with s splits and can be used to estimate the cost for the minimum cost subtree rooted at a node N. A point to note is that when the number of splits considered increases from s to $s+1$, the minimum subtree cost (as described in Theorem 5.1 increases by $2+\log a$ and decreases by n_{s+2}.

Thus, we simply need to compute, using the result of Theorem 5.1, the minimum cost for subtrees rooted at N with $0, \ldots, k-1$ splits and set our cost estimate to be the minimum of all these costs. The reason for only considering upto $k-1$ splits is that beyond $k-1$ splits, the subtree cost does not reduce any further. This is because the last term in Theorem 5.1 (which is the sum of the number of records of the $k-s-1$ smallest classes) becomes 0 for $k-1$ splits and cannot decrease any further, while the other terms keep increasing with the number of splits.

In addition, if for a certain number s of splits, it is the case that $n_{s+2} \leq 2+\log a$, then we do not need to consider subtrees with splits greater than s. The reason for this is that when s increases and becomes $s+1$, the minimum subtree cost increases by a fixed amount which is $2+\log a$ while it decreases by n_{s+2}. Thus, since $n_i \geq n_{i+1}$, increasing s further cannot cause the minimum subtree cost to decrease any further.

The algorithm for computing the estimate for the minimum cost subtree at a "yet to be expanded" node N in PUBLIC(S) is as shown in figure 6. In the procedure, the variable tmpCost stores the minimum cost subtree with $s \geq 1$ splits. For $s = 0$, since the bound due to Theorem 5.1 may be loose, the procedure uses $C(S)+1$ instead. The maximum value considered for s is $k-1$, and if for an s, $n_{s+2} \leq 2+\log a$, then values larger than s are not considered. The time complexity of the procedure is dominated by the cost of sorting the n_i's in the decreasing order of their values, and is thus, $O(k \log k)$.

5.2. *Incorporating costs of split values*

In the PUBLIC(S) algorithm described in the previous section, when estimating the cost of a subtree rooted at a "yet to be expanded" node N, we estimate the cost of each split to be $\log a$ bits. However, this only captures the cost of specifying the attribute involved in the

split. In order to completely describe a split, a value or a set of values is also required for the splitting attribute—this is to specify the distribution of records amongst the children of the split node.

In this section, we present the PUBLIC(V) algorithm, which estimates the cost of each split more accurately than PUBLIC(S) by also including the cost of encoding the split value in the cost of each split. Except for this, the PUBLIC(V) algorithm follows a similar strategy as PUBLIC(S) for estimating the cost of the cheapest subtree rooted at a "yet to be expanded" node N. For values of $s \geq 0$, PUBLIC(V) first computes a lower bound on the cost of subtrees containing s splits and rooted at N. The minimum of these lower bounds for the various values of s is then chosen as the cost estimate for the cheapest subtree at N. The time complexity of the cost estimation procedure for PUBLIC(V) is $O(k * (\log k + a))$.

For a subtree with s splits and rooted at a node N, the cost of specifying the structure of the subtree is $2 * s + 1$, while the cost of encoding the splitting attribute at the internal nodes is $s * \log a$. Compared to the cost for specifying the splitting attribute which requires a fixed number of bits, $\log a$, estimating the cost of split values is a difficult problem. The reason for this is that we are trying to compute a lower bound on the cost of a subtree with s splits and rooted at a "yet to be expanded" leaf node N. As a result, we do not know in advance the splitting attribute for a split node in the subtree—the cost of encoding the split value depends on the splitting attribute. Also, it is difficult to estimate the number of values for the splitting attribute at a split node since the distribution of records at the leaves and the structure of the subtree is unknown. In order to keep the minimum subtree cost estimation procedure computationally efficient, we make the assumption that all we know is that the subtree has s split nodes. In the following subsections, for a subtree with s splits, we compute a lower bound on the cost of encoding the split values at the internal nodes and the cost of describing the records in the leaves. After presenting the underlying intuition, We develop, in Section 5.2.2, the overall algorithm to compute the minimum subtree cost.

5.2.1. Intuition. Before we present an algorithm for computing a lower bound on the cost of encoding the split values and the records in the leaves, we need to first devise a cost model for these in a specific subtree with s splits and $s + 1$ leaves. This is later used to formulate the problem of finding a lower bound in a manner that is independent of the exact distribution of the records in the leaves and the internal structure of the tree (since this is unknown). Let S be the set of records in node N and k be the number of classes in N. Also, let S_i denote the set of records in S that belong to class i, $n_i = |S_i|$, and S_{ij}, the set of records belonging to class i in leaf j. Note that $|S_{ij}| \leq n_i$. Furthermore, let c_j be the majority class for leaf j and M be the set of majority classes in the $s + 1$ leaves.

Note that classes not contained in M are a minority in all the leaves. In addition, a class $i \in M$ is a minority in leaves j for which $c_j \neq i$. Thus, using arguments similar to those employed in the proof of Theorem 5.1, it can be shown that the cost of encoding the records in the leaves is at least

$$\sum_{i \notin M} n_i + \sum_{i \in M} n_i - \sum_{j=1}^{s+1} |S_{c_j j}| \tag{2}$$

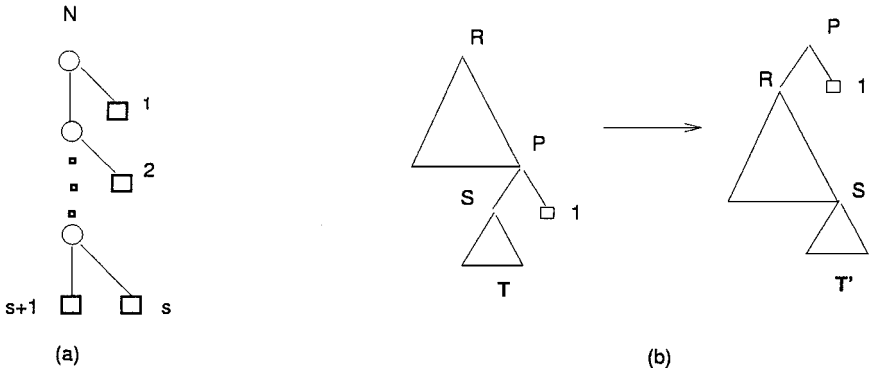


Figure 7. Subtree with minimum cost for encoding split values.

Next, we focus on the cost of encoding the split values. Suppose $V(S_{ij})$ denotes the minimum cost for encoding a split value at the parent of a node containing the set of records S_{ij}. We show later, how to estimate $V(S_{ij})$. A simple observation is that the set of records S_{ij} is contained in every ancestor of leaf j in the subtree. Thus, if leaf j is a descendant of an internal node, then the set of records S_{ij} occur in one of the internal node's children. Since leaf j contains the records in $S_{c_j j}$, it follows that the cost of specifying a split value at an internal node is greater than or equal to the maximum $V(S_{c_j j})$ for leaves j that are its descendants.

Since we are interested in computing a lower bound on the cost of specifying split values, we need to find the tree for which the sum of the costs of encoding split values at its internal nodes is minimum. Without loss of generality, suppose $V(S_{c_1 1}) \geq V(S_{c_2 2}) \geq \cdots \geq V(S_{c_{s+1} s+1})$. Then the tree with the least cost for encoding split values at internal nodes is as shown in figure 7(a) and the minimum cost of encoding splits is $\sum_{i=1}^{s} V(S_{c_i i})$. This can be shown using the following simple observation. Consider a subtree T with leaves numbered $1, \ldots, s+1$ where 1 is the leaf with the maximum value for $V(S_{c_i i})$. Let T' be the subtree obtained by deleting leaf 1 and its parent from T, and making leaf 1's parent the root of the resulting tree (see figure 7(b)). The cost of encoding split values for T' is no more than that for T. This is because the cost of representing the split value in every ancestor of 1 in T previously does not increases in T' while the cost for other internal nodes remains unchanged. Therefore, using induction with the above observation, we can show that the tree in figure 7(a) is a lower bound on the cost of representing split values for subtrees with leaves $1, \ldots, s+1$ where $V(S_{c_1 1}) \geq V(S_{c_2 2}) \geq \cdots \geq V(S_{c_{s+1} s+1})$.

Thus, in general, the cost of encoding the split values at all the internal nodes is at least $\sum_{j=1}^{s+1} V(S_{c_j j}) - \min\{V(S_{c_j j}) : 1 \leq j \leq s+1\}$. Combining the cost of specifying split values described above with the cost of encoding the records in the leaf nodes (see Eq. (2)), we obtain (after simplification) the total cost to be at least

$$\sum_{i=1}^{k} n_i - \left(\sum_{j=1}^{s+1} \left(|S_{c_j j}| - V(S_{c_j j}) \right) + \min \left\{ V(S_{c_j j}) : 1 \leq j \leq s+1 \right\} \right) \tag{3}$$

Thus, since we are interested in a lower bound on the total cost, we need to find a set M of majority classes (these are the c_j's for the $s+1$ leaves), and the set of records $S_{c_j j}$ for them such that the quantity in the following equation is maximized.

$$\sum_{j=1}^{s+1} \left(|S_{c_j j}| - V(S_{c_j j}) \right) + \min \left\{ V(S_{c_j j}) : 1 \leq j \leq s+1 \right\} \tag{4}$$

We still need to define $V(S_{ij})$ for a set of records S_{ij}. Let v_A denote the number of distinct values that occur for attribute A in records belonging to S_{ij}. We define $V(S_{ij})$ as follows.

$$V(S_{ij}) = \min_A \{ v : v = \log(v_A) \text{ if } A \text{ is numeric, else } v = v_A \text{ (if } A \text{ is categorical)} \} \tag{5}$$

We explain below our rationale for the above definition for $V(S_{ij})$. If A is the splitting attribute in the parent of a node that contains the set of records S_{ij}, then the number of values for attribute A in the parent node must be at least $v_A + 1$ (since the sibling node must contain at least one additional value). Thus, if A is numeric, the split value cost with attribute A as the splitting attribute is at least $\log(v_A)$ while if A is categorical, the cost is $\log(2^{v_A+1} - 2)$ or at least v_A since $\log(2^{v_A+1} - 2) \geq \log(2^{v_A})$ for all $v_A \geq 1$. As a result, in Eq. (5), since $V(S_{ij})$ is set to the minimum cost of split values over all the attributes A, the cost of specifying the split value at the parent of a node containing S_{ij} is at least $V(S_{ij})$.

Example 5.3. Consider the node N containing the 5 records described in Example 5.2. For our cost model, the minimum cost subtrees at N with 1 and 2 splits are as shown in figure 8(a) and (b), respectively. Each node has a cost of 1 for encoding its type. In addition, internal nodes have an additional cost of log 2 for specifying the splitting attribute. Furthermore, the second leaf node in figure 8(a) has an additional cost of at least 1 due to the record belonging to class medium. Also, for the subtree in figure 8(a), $c_1 =$ high and $c_2 =$ low. Thus, $S_{c_1 1}$ is the set of records with high class labels. We can compute $V(S_{c_1 1})$

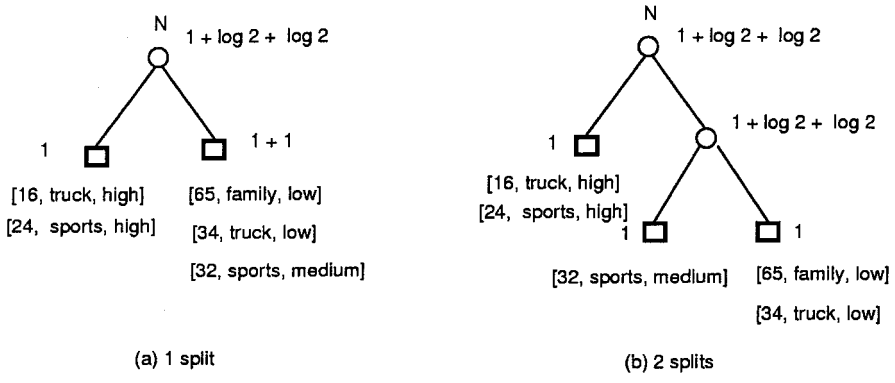


Figure 8. Minimum cost subtrees with 1 and 2 splits.

as follows. On both the *age* and the *car type* attributes, records in $S_{c_1 1}$ take 2 values. Since *age* is numeric and *car type* is categorical, $V(S_{c_1 1}) = \log 2$ which is the minimum of $\log 2$ and 2. Similarly, the value of $V(S_{c_2 2})$ can also be shown to be $\log 2$. Thus, the cost of encoding splits in the internal node for the subtree in figure 8(a) using our cost model is $1 + \log 2 + \log 2$.

The cost of each subtree is the sum of the costs for all the nodes. Thus, a lower bound on the cost of subtrees with 1 split is 6, while for subtrees with 2 splits, it is 9. These bounds are higher and thus more accurate than the bounds computed previously in Example 5.2 that only considered the cost of describing splitting attributes and not the cost of encoding split values.

The minimum cost of encoding the records and split values in a subtree with s splits can be found by computing the minimum possible value for Eq. (3). This is equivalent to computing a set M of majority classes such that Eq. (4) is maximized. We need to consider two possible cases—(1) each class is a majority class in at most one leaf, and (2) at least one class is a majority class in multiple leaves. Due to space constraints, we address only Case (1) in the body of this paper. Case (2) is dealt with in Appendix A.2.

For subtrees in which a class can be a majority class in at most one leaf, the majority classes c_j for the leaves are distinct and M contains $s + 1$ classes. The following lemma states the properties of classes in the set M that maximizes the value of Eq. (4).

Lemma 5.4. *For subtrees in which no two leaves contain the same majority class, for every class c_j in set M, the value of Eq.* (4) *is maximized when $S_{c_j j} = S_{c_j}$.*

Proof: See Appendix A.1. □

Thus, due to the above lemma, the optimal value for $S_{c_j j}$ is S_{c_j}, the set of all records belonging to class c_j in set S. As a result, for subtrees at node N with s splits and with no two leaves containing the same majority class, a lower bound on the cost of encoding split values and records in the leaves can be obtained by finding the set M containing $s + 1$ classes such that $\sum_{i \in M}(n_i - V(S_i)) + \min\{V(S_i) : i \in M\}$ is maximized.

5.2.2. Overall algorithm. Figure 9 illustrates the algorithm in PUBLIC(V) for computing the minimum cost for a subtree rooted at a "yet to be expanded" leaf node N and no two leaves of which have the same majority class. The estimate computed by PUBLIC(V) is at least as good as that computed by PUBLIC(S), and in many cases, better. For subtrees in which a class is a majority class in more than one leaf, the algorithm for computing a lower bound on the cheapest subtree is presented in Appendix A.2. The lower bound on the subtree cost is thus the minimum of the two computed lower bounds.

Due to Eq. (3), for the case of subtrees with s splits and each leaf with a different majority class, a lower bound on the cost is $2 * s + 1 + s * \log a + \sum_{i=1}^{k} n_i - (\sum_{i \in M}(n_i - V(S_i)) + \min\{V(S_i) : i \in M\})$, where M is a set of $s + 1$ classes for which $\sum_{i \in M}(n_i - V(S_i)) + \min\{V(S_i) : i \in M\}$ is maximized. Assuming that array A is as defined in Step 2 of the algorithm, in the following, we show that $\sum_{i=1}^{s}(n_{A[i]} - V(S_{L[i]})) + \max\{n_{A[i]} : s+1 \leq i \leq k\}$

procedure computeMinCostV(Node N):
1. **if** $k = 1$ **return** $(C(S) + 1)$
2. let array A contain the k classes in decreasing order of $n_i - V(S_i)$
3. s := 1
4. tmpCost := minCost := $1 + \sum_{i=1}^{k} n_i$
5. **while** $s < k$ **do** {
6. tmpCost := tmpCost + $2 + \log a - (n_{A[s]} - V(S_{A[s]}))$
7. minCost := $\min\{\text{minCost}, \text{tmpCost} - \max\{n_{A[i]} : s + 1 \leq i \leq k\}\}$
8. $s++$
9. }
10. **return** $\min\{C(S) + 1, \text{minCost}\}$

Figure 9. Algorithm for computing lower bound on subtree cost.

exceeds the maximum possible value for $\sum_{i \in M}(n_i - V(S_i)) + \min\{V(S_i) : i \in M\})$ and can thus be used in its place to compute the lower bound for the subtree with s splits.

A sketch of the proof is as follows. Consider the set M that maximizes $\sum_{i \in M}(n_i - V(S_i)) + \min\{V(S_i) : i \in M\})$. Let p be the class in M such that $n_p \leq n_i$ for all $i \in M$. First, since classes in A are sorted in the decreasing order of $n_i - V(S_i)$, it follows that $\sum_{i \in M-\{p\}}(n_i - V(S_i)) \leq \sum_{i=1}^{s}(n_{A[i]} - V(S_{A[i]}))$. Next, $n_p - V(S_p) + \min\{V(S_i) : i \in M\} \leq n_p$, and $n_p \leq \max\{n_{A[i]} : s + 1 \leq i \leq k\}$ since n_p is the $s + 1^{st}$ largest class in M. This concludes the sketch of our proof.

In the algorithm in figure 9, for a given s, tmpCost stores the quantity $2 * s + 1 + s * \log a + \sum_{i=1}^{k} n_i - \sum_{i=1}^{s}(n_{A[i]} - V(S_{L[i]}))$, and thus the minimum cost for a subtree with s splits can be obtained by subtracting $\max\{n_{A[i]} : s+1 \leq i \leq k\}$ from tmpCost. Note that we only need to contain subtrees with less than k splits, since for trees with k or greater splits, a pair of leaves would be forced to contain the same majority class. Thus, the minimum cost for a subtree can be found by computing minimum costs for subtrees with 1 through $k - 1$ splits and taking the lowest among them (variable minCost in the algorithm is used to keep track of this).

The complexity of the estimation procedure is dominated by the costs of computing $V(S_i)$ for the classes, and the cost of sorting array A. The $V(S_i)$'s for the k classes can be computed when attribute lists for a node are split and requires time $O(ka)$, while the sorting can be accomplished in time $O(k \log k)$. Note that the quantity $\max\{n_{A[i]} : s + 1 \leq i \leq k\}$ for each s can be precomputed for array A in $O(k)$ time by performing a single reverse scan of A.

6. Experimental results

In order to investigate the performance gains that can be realized due to PUBLIC's integrated approach to classification, we conducted experiments on real-life as well as synthetic data sets. We used an implementation of SPRINT (Shafer et al., 1996) as described in Section 3 as representative of traditional classifiers that carry out building and pruning in separate

phases. We are thus primarily interested in the speedup due to the integrated PUBLIC algorithms as measured against SPRINT.

Since real-life data sets are generally small, we also used synthetic data sets to study PUBLIC's performance on larger data sets. The purpose of the synthetic data sets is primarily to examine the PUBLIC's sensitivity to parameters such as noise, number of classes and number of attributes. Synthetic data sets allow us to vary the above parameters in a controlled fashion. Since PUBLIC is based on SPRINT except for the integration of the building and pruning phases, and SPRINT was shown to scale well for large databases in Shafer et al. (1996), our goal is not to demonstrate the scalability of PUBLIC. Instead, as we mentioned before, we are more interested in measuring the improvements in execution time due to combining the building and pruning phases compared to performing the two phases separately.

All of our experiments were performed using a Sun Ultra-2/200 machine with 512MB of RAM and running Solaris 2.5. Our experimental results with both real-life as well as synthetic data sets clearly demonstrate the effectiveness of PUBLIC's integrated algorithm compared to traditional classification algorithms.

6.1. Algorithms

In our experiments, we compared the execution times for four algorithms, whose characteristics we summarize below.

- **SPRINT:** This is the algorithm that we use as representative of traditional algorithms with separate building and pruning phases.
- **PUBLIC(1):** This is the simplest of the PUBLIC algorithms. It performs building and pruning together, and uses the very conservative estimate of 1 as the cost of the cheapest subtree rooted at a "yet to be expanded" leaf node.
- **PUBLIC(S):** Unlike PUBLIC(1), for the minimum cost subtree at a "yet to be expanded" leaf node, PUBLIC(S) considers subtrees with splits and includes the cost of specifying the splitting attribute for splits.
- **PUBLIC(V):** Among the PUBLIC algorithms, PUBLIC(V) computes the most accurate lower bound on the cost for a subtree at a "yet to be expanded" leaf node. In addition to the cost of specifying the splitting attribute for splits, it also considers the cost of specifying split values. Our implementation of PUBLIC(V) includes the portions described in Section A.3 Section A.4.

The integrated PUBLIC algorithms are implemented using the same code base as SPRINT except that they perform pruning while the tree is being built. The tree itself is built depth-first, and the pruning procedure is invoked on the tree each time a node is split.

6.2. Real-life data sets

We experimented with eight real-life datasets whose characteristics are illustrated in Table 2. These datasets were obtained from the UCI Machine Learning Repository.[2] Data sets in the UCI Machine Learning Repository often do not have both training and test data sets. For

Table 2. Real-life data sets.

Data set	Breast cancer	Car	Letter	Satimage	Shuttle	Vehicle	Yeast
No. of categorical attributes	0	6	0	0	0	0	0
No. of numeric attributes	9	0	16	36	9	18	8
No. of classes	2	4	26	7	5	4	10
No. of records (train)	469	1161	13368	4435	43500	559	1001
No. of records (test)	214	567	6632	2000	14500	287	483

these data sets, we randomly choose 2/3 of the data and used it as the training data set. The rest of the data is used as the test data set. Table 2 shows the number of records for both the training and the test data sets.

6.3. Results on real-life data sets

For each of the real-life data sets, we counted the number of nodes generated by each algorithm (see Table 3). The last but one row of Table 3 (labeled "Max Ratio") contains the percentage of additional nodes generated by SPRINT compared to PUBLIC(V), the best PUBLIC algorithm. The number of nodes in the final tree for each data set is contained in the final row of the table.

Intuitively, the number of nodes generated is a good measure of the work done by a classifier since decision tree classifiers spend most of their time splitting the generated nodes (more than 95%). From the table, it follows that for certain data sets (e.g., yeast), SPRINT may generate as many as 99% more nodes than PUBLIC(V). This confirms our conjecture that by pruning early, PUBLIC can result in a significant reduction in the number of redundant nodes generated. Note that even though PUBLIC can result in substantial decreases in the number of generated nodes compared to SPRINT, it still generates more nodes than the final number of nodes in the tree (after pruning). This suggests that there may still be room for further reducing the number of generated nodes.

Note that there is no direct corelation between the reduction in the number of generated nodes by PUBLIC and the data set size, the number of attributes or the number of classes

Table 3. Real-life data sets: Number of nodes generated.

Data set	Breast cancer	Car	Letter	Satimage	Shuttle	Vehicle	Yeast
SPRINT	21	97	3265	657	53	189	325
PUBLIC(1)	17	83	3215	565	53	141	237
PUBLIC(S)	15	71	2979	457	53	115	169
PUBLIC(V)	15	65	2875	435	53	107	163
Max ratio	40%	48%	14%	51%	0%	77%	99%
Nodes in final tree	9	37	1991	185	51	35	43

Table 4. Real-life data sets: Execution time (secs).

Data set	Breast cancer	Car	Letter	Satimage	Shuttle	Vehicle	Yeast
SPRINT	0.87	1.59	334.90	177.65	230.62	11.98	6.56
PUBLIC(1)	0.82	1.51	285.56	167.78	229.21	10.58	5.55
PUBLIC(S)	0.83	1.44	289.70	166.44	230.26	9.81	4.94
PUBLIC(V)	0.81	1.45	300.48	159.83	227.26	9.64	4.89
Max ratio	9%	0%	17%	11%	2%	2%	3%

in the data set. Rather, the effectiveness of PUBLIC depends more on the distribution of data and the degree of noise/outliers in the data set since these factors influence the number of nodes pruned in the final tree. In general, PUBLIC performs better when there is more opportunity for pruning in the data set. This is corroborated by the data in Table 3. For instance, in the yeast data set on which PUBLIC delivers the best results, nearly 100% of the nodes generated in the building phase by SPRINT are pruned in the pruning phase. In contrast, in the shuttle data set no nodes in the tree are pruned. Note that since the building phase of traditional classifiers split nodes until they become pure, the number of nodes pruned increases with the amount of noise in the underlying data. Thus, PUBLIC can be expected to result in better performance improvements for data sets with more noise.

We also present the execution times for the algorithms on the various data sets in Table 4. The final row indicates how much worse SPRINT is compared to the best PUBLIC algorithm. In general, PUBLIC results in improvements to execution times that are inferior compared to those realized for the number of nodes generated. There are two reasons for this. One is that the nodes that PUBLIC does not generate tend to be deeper in the tree and so contain fewer data records. As a result, the amount of work conserved for each node (in terms of scanning data for splits) that is pruned early is smaller compared to the amount of work already done for the ancestor nodes. Thus, the improvements in execution times are less than the improvements in the number of nodes generated. Also, note that compared to SPRINT which invokes the pruning procedure only once at the end, the PUBLIC algorithms may invoke the pruning procedure several times. Further, each invocation of the pruning procedure also computes estimates for the "yet to be expanded" leaves. However, our experience has been that the additional overhead of this is miniscule.

The smaller run time improvements with PUBLIC can also be attributed to the fact that the real-life data sets contain a few thousand records, and are thus are fairly small. As a result, the overhead of initializing the root node and sorting the attribute lists initially is significant relative to the time to construct the decision tree itself, and is common to both SPRINT and PUBLIC. Thus, if we exclude the preprocessing cost of building attribute lists from the execution times, the performance improvement due to PUBLIC is much better as illustrated in Table 5. Consequently, for large data sets, we expect the execution times for PUBLIC to be much better compared to SPRINT; this is corroborated by our results for synthetic data sets in the following subsection.

Table 5. Real-life data sets: Execution time without initial sorting overhead (secs).

Data set	Breast cancer	Car	Letter	Satimage	Shuttle	Vehicle	Yeast
SPRINT	0.6	1.27	304.04	150.07	176.16	10.44	6.01
PUBLIC(1)	0.55	1.18	254.62	139.95	174.88	9.02	5.0
PUBLIC(S)	0.56	1.11	256.30	137.15	175.77	8.28	4.39
PUBLIC(V)	0.54	1.12	269.53	132.36	173.82	8.11	4.34
Max Ratio	13%	13%	19%	13%	2%	39%	38%

6.4. *Synthetic data set*

In order to study the sensitivity of PUBLIC to parameters such as noise in a controlled environment, we generated synthetic data sets using the data generator used in Agrawal et al. (1993), Mehta et al. (1996) and Shafer et al. (1996) and available from the IBM Quest home page.[3] Every record in the data sets has nine attributes and a class label which takes one of two values. A description of the attributes for the records is as shown in Table 6. Among the attributes, elevel, car and zipcode are categorical, while all others are numeric. Different data distributions are generated by using one of ten distinct classification functions to assign class labels to records. Function 1 involves a predicate with ranges on a single attribute value. Functions 2 and 3 use predicates over two attributes, while functions 4, 5, 6 have predicates with ranges on three attributes. Functions 7 through 9 are linear functions and function 10 is a non-linear function (Agrawal et al., 1993). Further details on these ten predicates can be found in Agrawal et al. (1993). To model fuzzy boundaries between the classes, a perturbation factor for numeric attributes can be supplied to the data generator (Agrawal et al., 1993). In our experiments, we used a perturbation factor of 5%. We also varied the noise factor from 2 to 10% to control the percentage of noise in the data set. The number of records for each data set is set to 75000.

Table 6. Description of attributes in synthetic data sets.

Attribute	Description	Value
salary	Salary	Uniformly distributed from 20000 to 150000
commission	Commission	If salary $\geq$75000 then commission is zero else uniformly distributed from 10000 to 75000
age	Age	Uniformly distributed from 20 to 80
elevel	Education level	Uniformly chosen from 0 to 4
car	Make of the car	Uniformly chosen from 1 to 20
zipcode	Zip code of the town	Uniformly chosen from 9 to available zipcodes
hvalue	Value of the house	Uniformly distributed from $0.5k100000$ to $1.5k100000$ where $k \in \{0, \ldots, 9\}$ depends on zipcode
hears	Years house owned	Uniformly distributed from 1 to 30
loan	Total loan amount	Uniformly distributed from 0 to 500000

Table 7. Synthetic data sets: Number of nodes generated.

Predicate no.	1	2	3	4	5	6	7	8	9	10
SPRINT	20133	18819	19639	26823	17749	20069	18977	20583	18839	19255
PUBLIC(1)	7933	7175	7771	11173	7173	7879	8441	8287	7459	7287
PUBLIC(S)	6435	5887	6329	8947	5817	6435	6873	6667	6061	5927
PUBLIC(V)	5823	5391	5657	8105	5299	5811	6191	5979	5449	5363
Max ratio	246%	249%	245%	231%	235%	245%	207%	244%	246%	259%
Nodes in final tree	61	27	61	25	55	59	75	25	23	23

Table 8. Synthetic data sets: Execution time (secs).

Predicate no.	1	2	3	4	5	6	7	8	9	10
SPRINT	8519	3627	5919	15185	4343	12572	9142	6005	2098	4414
PUBLIC(1)	1493	1363	1427	1527	1350	1468	1306	1439	1529	1448
PUBLIC(S)	1428	1331	1413	1483	1306	1432	1271	1396	1501	1447
PUBLIC(V)	1415	1348	1428	1468	1290	1406	1253	1385	1475	1393
Max ratio	502%	169%	314%	934%	237%	974%	630%	334%	42%	217%

6.5. *Results on synthetic data sets*

In Tables 7 and 8, we present the number of nodes generated and the execution times for the data sets generated by functions 1 through 10. For each data set, the noise factor was set to 10%. From the tables, we can easily see that PUBLIC outperforms SPRINT by a significant amount. For example, SPRINT is more than 900% slower than PUBLIC(V) for functions 4 and 6, and more than 200% slower than PUBLIC(V) for most other functions. The wide variance in SPRINT's performance for the range of functions is due to differences in the structure of the trees generated for each function. In general, the construction of deeper highly skewed trees incurs more I/O than balanced shallow trees. It is interesting to observe that PUBLIC(1), the simplest of the PUBLIC algorithms, results in most of the realized gains in performance. The subsequent reductions in execution time due to PUBLIC(S) and PUBLIC(V) are not as high. Also, the time spent for generating attribute lists for the synthetic data sets was approximately 95 seconds. So, the improvement in running times excluding the cost of building attribute lists is slightly better.

We also performed experiments to study the effects of noise on the performance of PUBLIC. We varied noise from 2% to 10% for every function, and found that the execution of the algorithms on all the data sets were very similar. As a result, in figure 10 and figure 11, we only plot the execution times and number of generated nodes for functions 5 and 6. From the graphs, it follows that both execution times and the number of generated nodes increase as the noise is increased. This is because as the noise is increased, the size of the tree and thus the number of nodes generated increases. Furthermore, the running times for

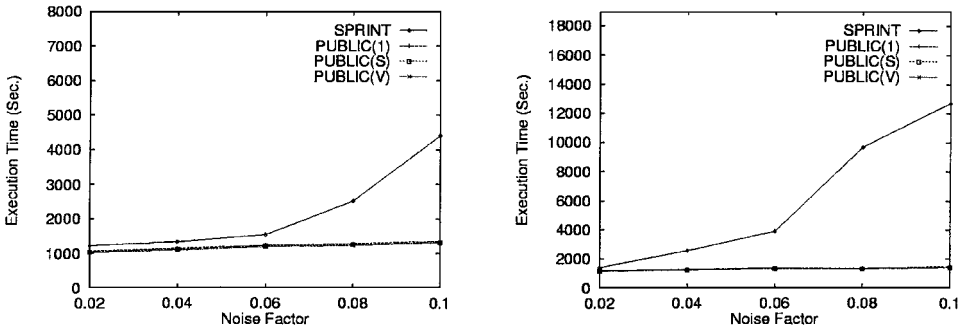


Figure 10. Synthetic data sets: Execution time (secs).

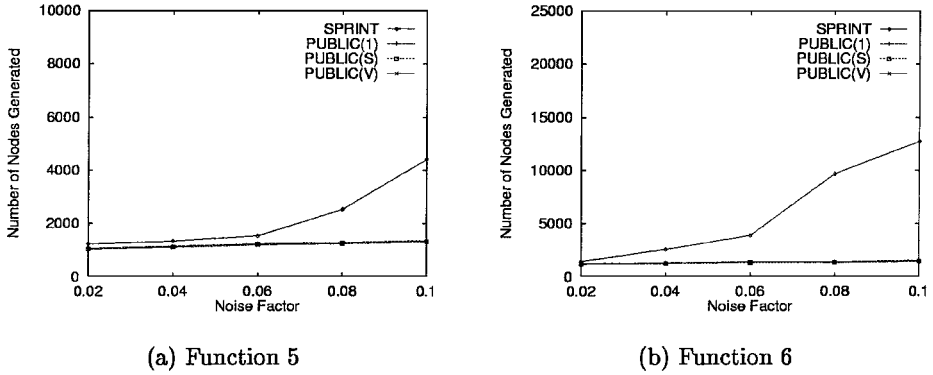
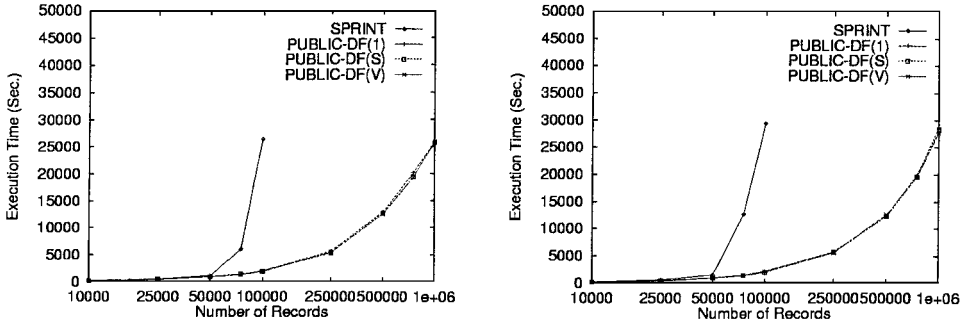


Figure 11. Synthetic data sets: Number of nodes generated.

SPRINT increase at a faster rate than those for PUBLIC as the noise factor is increased. Thus, PUBLIC results in better performance improvements at higher noise values. We also conducted experiments in which we varied the number of classes as well as the number of attributes in the data sets. However, we found that the performance of PUBLIC relative to SPRINT did not vary much for the different parameter settings.

The goal of our final set of experiments is to study the scale-up performance of PUBLIC. We varied the number of records in the training set from 10,000 to 1000,000 for every function, and found that the execution of the algorithms on all the data sets were very similar. As a result, in figure 12, we again plot the execution times for functions 5 and 6 only. From the graphs, we can see that the PUBLIC algorithms scale much better than SPRINT for large data sets. The execution times for SPRINT increase at a faster rate than those for PUBLIC as the number of records is increased. Consequently, we could not measure the running time for SPRINT beyond 100,000 records—this establishes the effectiveness of PUBLIC for handling large datasets.



(a) Function 5 (b) Function 6

Figure 12. Synthetic data sets: Execution time (secs).

7. Concluding remarks

Traditional decision tree classifiers generally construct a decision tree in two distinct phases. In the first *building* phase, a decision tree is first built by repeatedly scanning database, while in the second *pruning* phase, nodes in the built tree are pruned to improve accuracy and prevent overfitting. A drawback with performing the building and pruning actions in separate phases is that it could result in a fair amount of wasted effort since a significant portion of the tree generated during the building phase may subsequently be pruned during the pruning phase.

In this paper, we proposed a new classifier, PUBLIC, that integrates the pruning phase into the building phase. Specifically, nodes that are certain to be pruned are not expanded during the building phase—as a result, fewer nodes are expanded during the building phase, and thus the amount of work (e.g., disk I/O) required to construct the decision tree is reduced. In order to determine, during the building phase, nodes that are certain to be pruned, we need to know the cost of encoding the subtrees at the node. For this, we developed three techniques for computing a lower bound on the cost of a subtree at a "yet to be expanded" leaf node. By performing additional computation, each successive technique is able to generate more accurate estimates for the minimum cost subtree. Experimental results with real-life as well as synthetic data sets show that PUBLIC (with the tree estimation techniques for the minimum cost subtree) can result in significant performance improvements compared to traditional classifiers such as SPRINT.

Appendix

A. Estimating value costs in splits

A.1. Proofs of lemmas

Proof: (Lemma 5.4) Since S_{c_j} is the largest possible set for $S_{c_j j}$, $V(S_{c_j j})$ is maximum for $S_{c_j j} = S_{c_j}$. Thus, since the sets S_{c_j} are non-overlapping, the second term in Eq. (4) is

maximum when $S_{c_j j} = S_{c_j}$. For the first term in Eq. (4), we consider $(|S_{c_j j}| - V(S_{c_j j}))$ for each leaf j separately since the majority classes c_j for the leaves are different. Suppose $(|S_{c_j j}| - V(S_{c_j j}))$ is maximum for $S_{c_j j} = \hat{S}_{c_j}$, where $\hat{S}_{c_j} \subset S_{c_j}$. We show that this leads to a contradiction and thus, $(|S_{c_j j}| - V(S_{c_j j}))$ is maximum for $S_{c_j j} = S_{c_j}$. Let A be the attribute with the minimum split value cost for $\hat{S}_{c_j}$—thus, the split cost for A determines the value of $V(\hat{S}_{c_j})$.

Suppose A is a numeric attribute and $V(\hat{S}_{c_j}) = \log(v_A)$. Then, with $S_{c_j j}$ equal to S_{c_j} instead of $\hat{S}_{c_j}$ causes $(|S_{c_j j}| - V(S_{c_j j}))$ to increase by $(n_{c_j} - |\hat{S}_{c_j}|)$ and decrease by at most $\log(v_A + n_{c_j} - |\hat{S}_{c_j}|) - \log(v_A)$ (since at most $n_{c_j} - |\hat{S}_{c_j}|$ new values for attribute A can be introduced into set $S_{c_j j}$). Since $v_A \geq 1$ and for positive l, $l \geq \log(\frac{v_A + l}{v_A})$, it follows that $(|S_{c_j j}| - V(S_{c_j j}))$ is larger for $S_{c_j j} = S_{c_j}$ than for $S_{c_j j} = \hat{S}_{c_j}$.

On the other hand, if A were a categorical attribute and $V(\hat{S}_{c_j}) = v_A$, then with $S_{c_j j}$ equal to S_{c_j} instead of $\hat{S}_{c_j}$ causes $(|S_{c_j j}| - V(S_{c_j j}))$ to increase by $(n_{c_j} - |\hat{S}_{c_j}|)$ and decrease by at most $(v_A + n_{c_j} - |\hat{S}_{c_j}|) - v_A$. Consequently, $(|S_{c_j j}| - V(S_{c_j j}))$ is maximum when $S_{c_j j} = S_{c_j}$. □

A.2. Class majority in multiple leaves

For a subtree in which the majority class for two leaves j and k are identical, $S_{c_j} = S_{c_k}$. As a result, we cannot claim, as we did in Lemma 5.4, that Eq. (4) is maximum when $S_{c_j j} = S_{c_j}$. The reason for this is that $S_{c_j j}$ and $S_{c_k k}$ are disjoint since the same record cannot be in two leaves. Thus, when $S_{c_j j} = S_{c_j}$, $S_{c_k k} = \emptyset$ since $c_k = c_j$ which requires $S_{c_j j}$ and $S_{c_k k}$ to both be subsets of S_{c_j}.

Note that having a class be the majority class in multiple leaves (say leaves j and k) has the potential to improve the value of Eq. (4). be negligible or very small compared to $V(S_{c_j j} \cup S_{c_k k})$. Thus, splitting a class's records over multiple leaves may have advantages over concentrating all of them in a single leaf. However, in order to determine best value for Eq. (4) when a class can span over multiple $S_{c_j j}$s, we may need to consider several possible ways in which the class's records can be split into multiple sets, which can be computationally expensive.

In order to address this problem, for a leaf j such that the majority class in the leaf c_j is also a majority class in other leaves, we simply ignore $V(S_{c_j j})$ for the leaf—that is, we assume that $V(S_{c_j j})$ is 0. This has the following two implications. First, since at least one class is a majority class over multiple leaves, the term $\min\{V(S_{c_j j}) : 1 \leq j \leq s + 1\}$ in Eq. (4) becomes 0 and so we do not need to consider it. Second, for a class that is a majority class in multiple leaves j, the contribution to Eq. (4) is the sum of the sizes of the sets $S_{c_j j}$ for the leaves (since we ignore $V(S_{c_j j})$ for the leaves). Thus, the maximum contribution from such a class c_j to Eq. (4) is n_{c_j}—this happens when the union of the sets $S_{c_j j}$ for the leaves where it is a majority class is S_{c_j}, the set of all the records belonging to the class. Furthermore, the maximum benefit can be achieved by simply distributing all the class's records over 2 leaves—spreading the class's records across more leaves does not increase the value for Eq. (4). Finally, if a class is a majority class in only one leaf, say j, then as shown in the previous subsection, it's maximum contribution to Eq. (4) is $n_j - V(S_{c_j})$ (when $S_{c_j j} = S_{c_j}$).

```
procedure computeMinCostV2(Node N):
1.  if k = 1 return (C(S) + 1)
2.  for i := 1 to k do {
3.      B[2 * i − 1] := n_i − V(S_i)
4.      B[2 * i] := V(S_i)
5.  }
6.  sort array B in decreasing order of B[i]
7.  s :=1
8.  tmpCost := minCost := 1 + Σ_{i=1}^{k} n_i
9.  while s < 2 * k do {
10.   tmpCost := tmpCost + 2 + log a − B[s]
11.    minCost := min{minCost, tmpCost − B[s + 1]}
12.   s + +
13. }
14. return min{C(S) + 1, minCost}
```

Figure 13. Algorithm for estimating cost of cheapest subtree.

Thus, since the number of leaves in the subtree is $s + 1$, the problem is to find disjoint sets of classes M_1 (comprising of classes that are a majority in exactly one leaf) and M_2 (containing classes which are a majority in 2 leaves) that satisfy the following constraints:

1. $|M_1| + 2 * |M_2| \leq s + 1$.
2. $\sum_{i \in M_1}(n_i - V(S_i)) + \sum_{i \in M_2} n_i$ is maximized.

A.3. Overall algorithm

Figure 13 presents the algorithm in PUBLIC(V) for computing the minimum cost for a subtree rooted at a "yet to be expanded" leaf node N which A class is a majority class in more than one leaf.

For subtrees with s splits and containing at least a pair of leaves with the same majority class, a lower bound on the cost is $2 * s + 1 + s * \log a + \sum_{i=1}^{k} n_i - (\sum_{i \in M_1}(n_i - V(S_i)) + \sum_{i \in M_2} n_i)$, where M_1 and M_2 are sets of classes satisfying the two constraints mentioned in Section A.2. Assuming that array B for the various classes is as defined in Steps 2–5, in the following, we show that the sum of the $s + 1$ largest elements in B exceeds the maximum possible value for $\sum_{i \in M_1}(n_i - V(S_i)) + \sum_{i \in M_2} n_i$ and can thus be used in its place to compute the lower bound for the subtree with s splits.

A sketch of the proof is as follows. Let M_1 and M_2 be the set of classes that maximize $\sum_{i \in M_1}(n_i - V(S_i)) + \sum_{i \in M_2} n_i$. First, for every class i in M_1, $B[2 * i - 1] \geq n_i - V(S_i)$. Next, for all classes $i \in M_2$, $B[2 * i - 1] + B[2 * i] = n_i$. Thus, if for a class $i \in M_1$, $B[2 * i - 1]$ does not make it to the top $s + 1$ elements in B, or for class $i \in M_2$, one of $B[2 * i - 1]$ or $B[2 * i]$ are not in the top $s + 1$ elements in B, then it follows that the missing elements are smaller than every one of the top $s + 1$ elements in B. Thus, we are guaranteed that the sum of the top $s + 1$ elements in B is at least $\sum_{i \in M_1}(n_i - V(S_i)) + \sum_{i \in M_2} n_i$.

A.4. Further optimizations

For a class that is a majority class in only leaf j, we can estimate the maximum value of $|S_{c_j j}| - V(S_{c_j j})$ more accurately. The basic idea is that if attribute A assumes v values in set S, and also v values in set S_{ij}, then it cannot qualify to be a candidate for splitting the parent of a node containing the set of records S_{ij}. Thus, the definition of $V(S_{ij})$ in Eq. (5) must be modified to choose the minimum from among attributes who are candidates for splitting with S_{ij} and not all the attributes (an attribute is a candidate for splitting with S_{ij} if the number of values it assumes in S_{ij} is *strictly* less than the number of values for it in S).

The new definition for $V(S_{ij})$ has the implication that $|S_{c_j j}| - V(S_{c_j j})$ may not be maximum for $S_{c_j j} = S_{c_j}$. Instead, in order to find the $S_{c_j j}$ that maximizes $|S_{c_j j}| - V(S_{c_j j})$, we need to consider, for every attribute, the largest subset of S_{c_j} such that the attribute is a candidate for the subset. From amongst these subsets, the subset for which $|S_{c_j j}| - V(S_{c_j j})$ is maximum (with the new modified definition of $V(S_{ij})$) is the desired value for $S_{c_j j}$. For an attribute, if it S_{c_j} is not a candidate for splitting with S_{c_j}, then the largest subset of S_{c_j} for which the attribute is a candidate can be computed by deleting from S_{c_j}, records with the least frequently occurring value for the attribute in S_{c_j}.

Acknowledgments

We would like to thank Narain Gehani, Hank Korth and Avi Silberschatz for their encouragement. Without the support of Yesook Shim, it would have been impossible to complete this work. The work of Kyuseok Shim was partially supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

Notes

1. All logarithms in the paper are to the base 2.
2. Available at http://www.ics.uci.edu/~mlearn/MLRepository.html.
3. The URL for the page is http://www.almaden.ibm.com/cs/quest/demos.html.

References

Agrawal, R., Ghosh, S., Imielinski, T., Iyer, B., and Swami, A. 1992. An interval classifier for database mining applications. In Proc. of the VLDB Conference, Vancouver, British Columbia, Canada, August, pp. 560–573.

Agrawal, R., Imielinski, T., and Swami, A. 1993. Database mining: A performance perspective. IEEE Transactions on Knowledge and Data Engineering, 5(6):914–925.

Breiman, L., Friedman, J.H., Olshen, R.A., and Stone, C.J. 1984. Classification and Regression Trees. Belmont: Wadsworth.

Bishop, C.M. 1995. Neural Networks for Pattern Recognition. New York: Oxford University Press.

Cheeseman, P., Kelly, J., Self, M., Stutz, J., Taylor, W., and Freeman, D. 1988. AutoClass: A Bayesian classification system. In 5th Int'l Conf. on Machine Learning, June, Morgan Kaufman.

Fayyad, U. 1991. On the induction of decision trees for multiple concept learning. PhD Thesis, The University of Michigan, Ann Arbor.

Fayyad, U. and Irani, K.B. 1993. Multi-interval discretization of continuous-valued attributes for classification learning. In Proc. of the 13th Int'l Joint Conference on Artificial Intelligence, pp. 1022–1027.
Fukuda, T., Morimoto, Y., and Morishita, S. 1996. Constructing efficient decision trees by using optimized numeric association rules. In Proc. of the Int'l Conf. on Very Large Data Bases, Bombay, India.
Goldberg, D.E. 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Morgan Kaufmann.
Gehrke, J., Ramakrishnan, R., and Ganti, V. 1998. Rainforest—A framework for fast decision tree classification of large datasets. In Proc. of the VLDB Conference, August, New York City, NY.
Hunt, E.B., Marin, J., and Stone, P.J. (Eds.) 1966. Experiments in Induction. New York: Academic Press.
Krichevsky, R. and Trofimov, V. 1981. The performance of universal encoding. IEEE Transactions on Information Theory, 27(2):199–207.
Mehta, M.P., Agrawal, R., and Rissanen, J. 1996. SLIQ: A fast scalable classifier for data mining. In EDBT 96, March, Avignon, France.
Mehta, M., Rissanen, J., and Agrawal, R. 1995. MDL-based decision tree pruning. In Int'l Conference on Knowledge Discovery in Databases and Data Mining (KDD-95), August, Montreal, Canada.
Mitchie, D., Spiegelhalter, D.J., and Taylor, C.C. 1994. Machine Learning, Neural and Statistical Classification. Ellis Horwood.
Quinlan, J.R. and Rivest, R.L. 1989. Inferring decision trees using minimum description length principle. Information and Computation, 30(3):227–248.
Quinlan, J.R. 1986. Induction of decision trees. Machine Learning, 1:81–106.
Quinlan, J.R. 1987. Simplifying decision trees. Journal of Man-Machine Studies, 27:221–234.
Quinlan, J.R. 1993. C4.5: Programs for Machine Learning. Morgan Kaufman.
Ripley, B.D. 1996. Pattern Recognition and Neural Networks. Cambridge: Cambridge University Press.
Rissanen, J. 1978. Modeling by shortest data description. Automatica, 14:465–471.
Rissanen, J. 1989. Stochastic Complexity in Statistical Inquiry. World Scientific Publ. Co.
Shafer, J., Agrawal, R., and Mehta, M. 1996. SPRINT: A scalable parallel classifier for data mining. In Proc. of the VLDB Conference, September, Bombay, India.
Wallace, C.S. and Patrick, J.D. 1993. Coding decision trees. Machine Learning, 11:7–22.
Zihed, D.A., Rakotomalala, R., and Feschet, F. 1997. Optimal multiple intervals discretization of continuous attributes for supervised learning. In Int'l Conference on Knowledge Discovery in Databases and Data Mining (KDD-97).