Data Mining and Knowledge Discovery, 7, 187–214, 2003
© 2003 Kluwer Academic Publishers. Manufactured in The Netherlands.

Building Decision Trees with Constraints

MINOS GAROFALAKIS minos@bell-labs.com
Bell Labs, Lucent Technologies, Murray Hill, NJ 07974, USA

DONGJOON HYUN hyundong@organ.kaist.ac.kr
Korea Advanced Institute of Science and Technology and Advanced Information Technology Research Center, Taejon, Korea

RAJEEV RASTOGI rastogi@bell-labs.com
Bell Labs, Lucent Technologies, Murray Hill, NJ 07974, USA

KYUSEOK SHIM* shim@ee.snu.ac.kr
Seoul National University and Advanced Information Technology Center, Seoul, Korea

Editors: Fayyad, Mannila, Ramakrishnan

Received November 30, 2000; Revised November 21, 2001

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. Frequently, however, the constructed trees are complex with hundreds of nodes and thus difficult to comprehend, a fact that calls into question an often-cited benefit that decision trees are easy to interpret. In this paper, we address the problem of constructing "simple" decision trees with few nodes that are easy for humans to interpret. By permitting users to specify *constraints* on tree size or accuracy, and then building the "best" tree that satisfies the constraints, we ensure that the final tree is both easy to understand and has good accuracy. We develop novel *branch-and-bound* algorithms for pushing the constraints into the building phase of classifiers, and pruning early tree nodes that cannot possibly satisfy the constraints. Our experimental results with real-life and synthetic data sets demonstrate that significant performance speedups and reductions in the number of nodes expanded can be achieved as a result of incorporating knowledge of the constraints into the building step as opposed to applying the constraints after the entire tree is built.

Keywords: data mining, classification, decision tree, branch-and-bound algorithm, constraint

1. Introduction

Background and motivation. Classification is an important problem in data mining. Under the guise of *supervised learning*, classification has been studied extensively by the AI community as a possible solution to the "knowledge acquisition" or "knowledge extraction" problem. Briefly, the input to a classifier is a *training set* of records, each of which is a tuple

*To whom correspondence should be addressed.

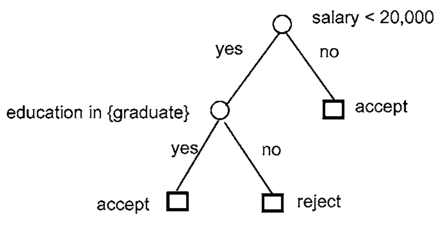
of *attribute* values tagged with a *class label*. A set of attribute values defines each record. Attributes with discrete domains are referred to as *categorical*, while those with ordered domains are referred to as *numeric*. The goal is to induce a concise model or description for each class in terms of the attributes. The model is then used to classify (i.e., assign class labels to) future records whose classes are unknown.

Classification has been successfully applied to wide range of application areas, such as medical diagnosis, weather prediction, credit approval, customer segmentation, and fraud detection. Many different techniques have been proposed for classification, including Bayesian classification (Cheeseman et al., 1988), neural networks (Bishop, 1995; Ripley, 1996), genetic algorithms (Goldberg, 1989) and tree-structured classifiers (Breiman et al., 1984). Among these proposals, *decision tree classifiers* (Murthy, 1998) have found the widest applicability in large-scale data mining environments. There are several reasons for this. First, compared to neural networks or a Bayesian classifiers, decision trees offer a very intuitive representation that is easy to assimilate and translate to standard database queries (Agrawal et al., 1992; Breiman et al., 1984). Second, while training neural networks can take large amounts of time and thousands of iterations, decision tree induction is efficient and is thus suitable for large training sets. Furthermore, 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 for the data or class labels) (Fayyad, 1991). Finally, the accuracy of decision tree classifiers is comparable or even superior to that of other classification techniques (Mitchie et al., 1994). The work reported in this paper focuses on decision tree classifiers.

Example 1.1. 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 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.

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.

Figure 1(b) depicts a decision tree for our example training data. 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), PUBLIC (Rastogi and Shim, 1998), BOAT (Gehrke et al., 1999)). Most of these algorithms consist of 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 (i.e., the partition is *pure*). 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 tree that is "perfect," in the sense that it 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" of the training data and to obtain a tree with higher accuracy. A number of pruning strategies have been proposed in the literature including MDL pruning (Fayyad and Irani, 1993; Mehta et al., 1995; Quinlan and Rivest, 1989; Wallace and Patrick, 1993), cost-complexity pruning (Quinlan, 1987), and pessimistic pruning (Quinlan, 1987).

Even after pruning, the decision tree structures induced by existing algorithms can be extremely complex, comprising hundreds or thousands of nodes and, consequently, very difficult to comprehend and interpret. The situation is only exacerbated by the voluminous, high-dimensional training data sets that are characteristic of modern decision support applications. This is a serious problem and calls into question an often-cited benefit of decision trees, namely that they are easy to assimilate by humans. In many scenarios, users are only interested in obtaining a "rough picture" of the patterns in their data; for example, during an *interactive data mining and exploration* session, a user is primarily interested in obtaining a fast, concise (and reasonably accurate) decision tree model to quickly identify the key patterns/rules in the underlying data. In such scenarios, users may actually find a simple, comprehensible, but only approximate decision tree much more useful than an accurate (e.g., MDL-optimal) tree that involves a lot of detail. The idea of simple, approximate decision trees becomes even more attractive by the fact that the size and accuracy of a decision tree very often follow a law of "*diminishing returns*"—for many real-life data sets, adding more nodes to the classifier results in monotonically decreasing gains in accuracy. As a consequence, in many situations, a small decrease in accuracy is accompanied by a dramatic reduction in the size of the tree. For example, Bohanec and Bratko (1994) consider a decision tree for deciding the legality of a white-to-move position in chess. They demonstrate that while a decision tree with 11 leaves is completely accurate, a subtree with only 4 leaves is 98.45% accurate, and a subtree with only 5 leaves is 99.57% accurate. Thus, more than half the size of the accurate tree accounts for less than 0.5% of the accuracy!

Our contributions. In this paper, we attempt to remedy the aforementioned problem by developing novel algorithms that allow users to effectively trade accuracy for simplicity during the decision tree induction process. Our algorithms give users the ability to specify *constraints* on either (1) the *size* (i.e., number of nodes); or, (2) the *inaccuracy* (i.e., MDL cost (Mehta et al., 1995; Quinlan and Rivest, 1989; Rastogi and Shim, 1998) or number of misclassified records) of the target classifier, and employ these constraints to *efficiently* construct the "best possible" decision tree. More specifically, let T denote the "accurate" decision tree built during traditional decision tree induction. Our work addresses the following two constrained induction problems:

(1) *Size-constrained decision trees.* Given an upper bound k on the size (i.e., number of nodes) of the classifier, build an *accuracy-optimal* subtree of T with *at most* k nodes; that is, build the subtree of T with size at most k that minimizes either (a) the total MDL cost, or (b) the total number of misclassified records.
(2) *Accuracy-constrained decision trees.* Given an upper bound $\mathcal{C}$ on the inaccuracy (total MDL cost or number of misclassified records) of the classifier, build a *size-optimal* subtree of T with inaccuracy *at most* $\mathcal{C}$; that is, build the smallest subtree of T whose total MDL cost or number of misclassified records does not exceed $\mathcal{C}$.

Thus, our constraint-based framework enables the efficient induction of decision tree classifiers that are simple and easy to understand, and, at the same time, have good accuracy characteristics.

A naive approach to building the desired optimal subtree that satisfies the user-specified size or accuracy constraint is to first grow the full (accurate) tree T, and then employ algorithms based on *dynamic programming* to prune away suboptimal portions of T until the constraint is satisfied. We propose time- and memory-efficient dynamic programming algorithms for pruning T to optimal size- and accuracy-constrained subtrees. Similar algorithms for the optimal pruning of accurate decision trees have also been proposed in the earlier work of Bohanec and Bratko (1994) and Almuallim (1996), where the accuracy measure was assumed to be the number of misclassified training set records (i.e., the "resubstitution error" of Breiman et al. (1984)). Our algorithms extend that earlier work by also considering the MDL cost of a subtree. Furthermore, our dynamic programming algorithms are much more efficient than those of Bohanec and Bratko and conceptually much simpler than the "left-to-right" dynamic programming procedure of Almuallim.

The problem with such naive approaches is that they essentially apply the size/accuracy constraints as an afterthought, i.e., after the complete decision tree has been built. Obviously, this could result in a substantial amount of wasted effort since an entire subtree constructed in the building phase may later be pruned when size/accuracy constraints are enforced. If, during the building phase, it is possible to determine that certain nodes will be pruned during the subsequent constraint-enforcement phase, then we can avoid expanding the subtrees rooted at these nodes. Since building a subtree typically requires repeated scans to be performed over the data, significant reductions in I/O and improvements in performance can be realized.

The major contribution of our work is the development of novel decision tree induction algorithms that *push size and accuracy constraints into the tree-building phase*. Our algorithms employ *branch-and-bound* techniques to identify, during the growing of the decision tree, nodes that cannot possibly be part of the final constrained subtree. Since such nodes are guaranteed to be pruned when the user-specified size/accuracy constraints are enforced, our algorithms stop expanding such nodes early on. Thus, our algorithms essentially integrate the constraint-enforcement phase into the tree-building phase instead of performing one after the other. Furthermore, by only pruning nodes that are guaranteed not to belong to the optimal constrained subtree, we are assured that the final (sub)tree generated by our integrated approach is *exactly the same* as the subtree that would be generated by a naive approach that enforces the constraints only after the full tree is built. Determining, during the building phase, whether a node will be pruned by size or accuracy constraints is problematic, since the decision tree is only partially generated. To guarantee that only suboptimal parts of the tree are pruned, requires us to estimate, at each leaf of the partial tree, a *lower bound* on the inaccuracy (MDL cost or number of misclassifications) of the subtree rooted at that leaf (based on the corresponding set of training records). Our branch-and-bound induction algorithms apply adaptations of our earlier results (Rastogi and Shim, 1998) on estimating such lower bounds to the problem of constructing size/accuracy-constrained decision trees. Our experimental results on real-life as well as synthetic data sets demonstrate that our approach of pushing size and accuracy constraints into the building phase can result in dramatic performance improvements compared to the naive approach of enforcing the constraints only after the full tree is built. The performance speedups, in some cases, can be as high as two or three orders of magnitude.

Roadmap. The remainder of this paper is organized as follows. Section 3 provides an overview of the building and pruning phases of a traditional decision tree classifier along the lines of SPRINT (Shafer et al., 1996) and CART (Breiman et al., 1984). In Section 4, we describe our dynamic programming algorithms for the naive solution of optimally pruning a fully-grown decision tree. Section 5 presents our integrated decision tree construction algorithms that push size and accuracy constraints into the tree-building phase. In Section 6, we discuss the findings of an extensive experimental study of our constrained decision tree induction algorithms using both synthetic and real-life data sets. Finally, Section 7 concludes the paper.

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 more recent work, Gehrke et al. (1998) propose a unifying framework, termed RainForest, for scaling up existing decision-tree algorithms. The basic idea of the RainForest framework lies in the use of sufficient statistics (termed AVC-sets) for top-down decision-tree induction; this guarantees scalable versions of existing decision-tree algorithms without modifying the final result. An optimistic approach to decision-tree induction, termed BOAT, has also been proposed by Gehrke et al. (1999). BOAT exploits statistical techniques to construct an initial tree based on only a small subset of the data and refine it to arrive at the final decision tree. As a consequence, BOAT can build several levels of the tree in only two scans over the training data, resulting in substantial performance gains over previous algorithms. None of these earlier proposals has considered the problem of building decision-tree classifiers in the presence of size or accuracy constraints.

An important class of pruning algorithms are those based on the Minimum Description Length (MDL) principle (Fayyad and Irani, 1993; Mehta et al., 1995; Quinlan and Rivest, 1989; Wallace and Patrick, 1993). 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.

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. In Fayyad and Irani (1993) and 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.

The PUBLIC decision-tree construction algorithm proposed in Rastogi and Shim (1998) integrates the pruning phase into the building phase. During building, PUBLIC employs a lower bound on the MDL cost of a node to detect if the node is guaranteed to be pruned during MDL pruning. If that is the case, PUBLIC prunes the node and stops expanding the decision tree in that direction. As a consequence, PUBLIC achieves substantial reduction in I/O costs compared to traditional decision-tree induction algorithms. Furthermore, by using a guaranteed lower bound on the MDL cost, PUBLIC guarantees that the tree generated by its integrated approach is exactly the same as the tree that would have been generated as a result of executing building and pruning sequentially, one after the other.

Bohanec and Bratko (1994) and Almuallim (1996) propose algorithms based on dynamic programming that, given a fully-grown, accurate decision tree, find optimal subtrees that satisfy a user-specified constraint on either size (i.e., number of nodes) or accuracy (i.e., number of misclassifications). Our algorithms extend that earlier work by also considering the MDL cost of a subtree and are much more efficient than those of Bohanec and Bratko, and conceptually much simpler than the "left-to-right" dynamic programming procedure of Almuallim. Most importantly, both of these earlier proposals enforce the constraints *only after the full decision tree has been built*. Obviously, this could result in a substantial amount of wasted effort since an entire subtree constructed in the building phase may later be pruned when size/accuracy constraints are enforced. Our algorithms, on the other hand, adapt an integrated approach based on the PUBLIC framework and are thus much more efficient.

3. Preliminaries

In this section, we present a brief overview of the building and pruning phases of a traditional decision tree classifier. More detailed descriptions of existing decision tree induction

```
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 node $N$ is not pure {
6.          for each attribute $A$
7.              Evaluate splits on attribute $A$
8.          Use best split to split $N$ into $N_1$ and $N_2$
9.          Append $N_1$ and $N_2$ to $Q$
10.     }
11. }
```

(a)

```
procedure PruneTree(Node $N$):
1. if $N$ is a leaf return $(C(S) + 1)$
2. minCost$_1$ := PruneTree($N_1$);
3. minCost$_2$ := PruneTree($N_2$);
4. minCost$_N$ := min{$C(S) + 1$ ,
           $C_{split}(N) + 1 +$ minCost$_1$ + minCost$_2$};
5. if minCost$_N$ = $C(S) + 1$
6.     prune child nodes $N_1$ and $N_2$ from tree
7. return minCost$_N$
```

(b)

Figure 2. (a) Tree-building algorithm; (b) Tree-pruning algorithm.

algorithms can be found in earlier literature (Breiman et al., 1984; Gehrke et al., 1998; Murthy, 1998; Rastogi and Shim, 1998; Shafer et al., 1996).

3.1. *Tree-building phase*

The overall algorithm for growing a decision tree classifier is depicted in figure 2(a). Basically, the tree is built breadth-first by recursively partitioning the data until each partition is *pure* (i.e., it only contains records belonging to the same class). The *splitting condition* for partitioning the data is of the form (1) $A < v$, if A is a numeric attribute (v is a value in the domain of A); or, (2) $A \in V$, if A is a categorical attribute (V is a set of values from A's domain). Thus, each split is binary.[1]

The splitting condition for each internal node of the tree is selected so that it minimizes an *impurity function*, such as the *entropy*, of the induced data partitioning (Breiman et al., 1984). (We should, of course, note that there exist split-attribute and split-point selection methods that are not impurity-based, e.g., methods based on the Chi-squared or the G-squared test (Murthy, 1998).) 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 is the one that provides the best separation among classes, and is thus chosen as the best split for a node. An alternative impurity function for choosing splitting conditions is the *Gini index* (Shafer et al., 1996). For a set S of records, the Gini index is defined as $1 - \sum_j p_j^2$, where p_j is the relative frequency of class j in S.

3.2. *Tree-pruning phase*

To prevent overfitting of the training data, the MDL principle (Rissanen, 1978, 1989) is applied to prune the tree built in the growing phase and make it more general. Briefly, the

Table 1. Notation.

Symbol	Description
T	Full tree constructed at the end of the building phase
T_p	Partially-built tree at some stage of the building phase
T_f	Final subtree of T (satisfying the user-specified constraints)
R	Root of tree constructed during the building phase
a	Number of attributes
N	Generic node of the decision tree
S	Set of records in node N
N_1, N_2	Children of node N
$C(S)$	Cost of encoding the classes for records in S
$C_{split}(N)$	Cost of encoding the split at node N
k	Constraint on the number of nodes in the final tree
$\mathcal{C}$	Constraint on the MDL cost/number of misclassified records in the final tree

MDL principle states that the "best" tree is the one that can be encoded using the smallest number of bits. Thus, the goal of the tree-pruning phase is to find the subtree of the tree grown in the tree-building phase that can be encoded with the least number of bits.

In what follows, we first present a scheme for encoding decision trees. We then describe a pruning algorithm that, in the context of our encoding scheme, finds the minimum MDL-cost subtree of the tree constructed in the growing phase. Table 1 summarizes some of the notation used throughout this paper; additional notation will be introduced when necessary.

Cost of encoding data records. Let a set S contain n records each belonging to one of c 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[2]

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

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.

Cost of encoding the tree. The cost of encoding the tree comprises three distinct components: (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; and, (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, this encoding technique for tree structures is near-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). In the rest of the paper, we refer to the cost of encoding a tree computed above as the *MDL cost* of the tree.

Pruning algorithm. As explained earlier, the goal of the pruning phase is to compute the minimum MDL-cost subtree of the tree T constructed in the building phase. Briefly, this is achieved by traversing T in a bottom-up fashion, pruning all descendents of a node N if the cost of the minimum-cost subtree rooted at N is greater than or equal to $C(S) + 1$ (i.e., the cost of directly encoding the records corresponding to N). The cost of the minimum-cost subtree rooted at N is computed recursively as the sum of the cost of encoding the split and structure information at N ($C_{split}(N) + 1$) and the costs of the cheapest subtrees rooted at its two children. Figure 2(b) gives the pseudocode for the pruning procedure; more details can be found in Rastogi and Shim (1998).

4. The naive approach: Constraint-based decision tree pruning

The pruning phase described in the previous section computes the subtree of T with minimum MDL cost (recall that T is the *complete* tree constructed during the tree-building phase). In this section, we consider more general constraints on the size and accuracy of the desired subtree of T, and develop algorithms for computing the subtree of T that satisfies these constraints.[3] The motivation for employing these richer constraints is to allow users to obtain approximate decision tree classifiers that are simple, easy to comprehend and, at the same time, fairly accurate. More specifically, we consider the following constraints on the desired subtree T_f of T:

1. *Size constraint.* For a given k, T_f contains at most k nodes and has the minimum possible MDL cost (or number of misclassified records).
2. *Accuracy constraint.* For a given $\mathcal{C}$, T_f has an MDL cost (or number of misclassified records) at most $\mathcal{C}$ and has the minimum possible number of nodes.

The number of misclassified records (i.e., the "resubstitution error" of Breiman et al. (1984) is, in some respect, a measure of the accuracy of the decision tree based on the given training set. Specifically, it is the sum of the number of misclassified training records in each leaf of the tree. A record is said to be misclassified if its class label differs from the label for the leaf (which is essentially the majority class in the corresponding set of records). SPRINT (Shafer et al., 1996) actually uses the number of misclassified training records in each leaf as the cost of encoding the corresponding set of data records.

Note that our size and accuracy constraints are fairly general and can, in fact, be used to specify the final tree computed by the pruning step of a traditional classifier (as described in Section 3). For instance, if we choose k to be very large, then the subtree T_f of T that satisfies our size constraint is identical to the subtree constructed by a classifier like PUBLIC (Rastogi and Shim, 1998) which uses MDL pruning. Thus, our constraints capture the two important dimensions of decision trees—size and accuracy.

In the following subsections, we present algorithms for computing the optimal size- or accuracy-constrained subtree T_f of T. Like the earlier work of Bohanec and Bratko (1994) and Almuallim (1996), our "naive approach" algorithms are based on dynamic programming and assume an input that consists of the complete tree T and the constraint (k or $\mathcal{C}$). Our work, however, also considers the MDL-cost formulation of the accuracy constraint, whereas their work focuses on only the number of misclassifications. Further, our algorithms are much more efficient than that of Bohanec and Bratko (1994) (which is quadratic in the size of T) and conceptually much simpler than the "left-to-right" dynamic programming procedure of Almuallim (which has time and memory demands similar to our algorithms).

4.1. Computing an optimal size-constrained subtree

Minimum MDL cost. Our algorithm for finding the minimum MDL-cost subtree of T with at most k nodes consists of two distinct steps. First, procedure COMPUTECOST (depicted in figure 3(a)) is executed to compute the minimum MDL-cost subtrees rooted at nodes of T. Second, procedure PRUNETOSIZEK (depicted in figure 3(b)) is run to prune suboptimal portions of T so that the final subtree T_f contains at most k nodes.

Procedure COMPUTECOST employs dynamic programming to compute (in Tree[N, l].cost) the cost of the minimum-cost subtree rooted at node N and containing at most l nodes. The key idea here is that Tree[N, l].cost is the minimum of (1) $C(S) + 1$, i.e., the cost of directly encoding records in N; and, (2) for $1 \leq k_1 \leq l - 2$, $C_{split}(N) + 1 +$ Tree[N_1, k_1].cost + Tree[N_2, $l - k_1 - 1$].cost, i.e., the cost of splitting node N plus the costs of the minimum-cost subtrees rooted at N's children, where the total number of nodes in these subtrees does not exceed $l - 1$. Procedure COMPUTECOST is initially invoked with the root node R of the full tree T and the constraint k on the number of nodes.

After procedure COMPUTECOST completes execution, Tree[N, l].left stores the size of the left subtree in the minimum cost subtree of size at most l rooted at N. Note that if

```
procedure ComputeCost(Node $N$, integer $l$):
1.  if Tree[$N$,$l$].computed = true
2.      return Tree[$N$,$l$].cost
3.  Tree[$N$,$l$].left := 0
4.  if $l < 3$ or $N$ is a leaf
5.      Tree[$N$,$l$].cost := $C(S) + 1$
6.  else {
7.      Tree[$N$,$l$].cost := $C(S) + 1$
8.      for $k_1$ := 1 to $l - 2$ do {
9.          $k_2 := l - k_1 - 1$
10.         Cost := $C_{split}(N) + 1+$
                $+$ ComputeCost($N_1$, $k_1$) $+$
                $+$ ComputeCost($N_2$, $k_2$)
11.         if Cost < Tree[$N$,$l$].cost {
12.             Tree[$N$,$l$].cost := Cost
13.             Tree[$N$,$l$].left := $k_1$
14.         }
15.     }
16. }
17. Tree[$N$,$l$].computed := true
18. return Tree[$N$,$l$].cost
```

(a)

```
procedure PruneToSizeK(Node $N$, integer $l$):
1.  if $N$ is a leaf return
2.  if $l < 3$ or Tree[$N$,$l$].left = 0
3.      Prune nodes $N_1$ and $N_2$
        (and their descendants)
4.  else {
5.      $k_1$ := Tree[$N$,$l$].left
6.      $k_2 := l - k_1 - 1$
7.      PruneToSizeK($N_1$, $k_1$)
8.      PruneToSizeK($N_2$, $k_2$)
9.  }
10. return
```

(b)

Figure 3. Algorithms for (a) Computing minimum MDL-cost subtrees (b) Pruning suboptimal subtrees.

Tree[N, l].left $= 0$, then the minimum cost subtree rooted at N with maximum size l consists of only the node N. Procedure PruneToSizeK is then called to prune nodes from T that do not belong to the minimum MDL-cost subtree of size at most l. It is invoked with R and k as the input parameters.

Minimum number of misclassified records. Algorithm ComputeCost can be easily modified to compute the subtree T_f with the minimum number of misclassified records and size at most k. The reason for this is that the dynamic programming relationship for minimizing MDL cost also holds for minimizing the number of misclassified records. More specifically, if Tree[N, l].cost stores the minimum number of misclassified records among all the subtrees rooted at N, then Tree[N, l].cost is the minimum of (1) the number of misclassified records in N, and (2) for $1 \leq k_1 \leq l-2$, Tree[N_1, k_1].cost + Tree[$N_2, l-k_1-1$].cost. Thus, to minimize the number of misclassifications, ComputeCost simply needs to be modified as follows. First, instead of setting Tree[N, l].cost to $C(S) + 1$ in steps 5 and 7, we simply set Tree[N, l].cost to the number of misclassified records in node N. Second, we delete the term $C_{split}(N) + 1$ from the right-hand side of the assignment statement in Step 10.

Time and space complexity. The time complexity of ComputeCost is $O(nk)$, where n is the number of nodes in T and k is the upper bound on the desired tree size. This is because for a node N and $l < k$, ComputeCost can be invoked at most $k - l$ times, once from each invocation of ComputeCost with N's parent and one of $l + 1, \ldots, k$. Procedure

PRUNETOSIZEK can be shown to have time complexity $O(n)$ since it visits each node at most once. Thus, the overall time complexity for computing the minimum cost subtree is $O(nk)$. The space complexity of the procedure is also $O(nk)$, since it only needs to store (in Tree[N, l]) the cost and left-subtree size for the "best" subtree rooted at node N with size at most $l \leq k$.

4.2. Computing an optimal accuracy-constrained subtree

We now consider the problem of computing the subtree of T with the minimum number of nodes and MDL cost (or number of misclassified records) at most $\mathcal{C}$. Our proposed solution works by invoking procedure COMPUTECOST on the root nodes R of T with increasing bounds on the number of nodes l (beginning with $l = 1$ and considering increments of 1), until an l is reached for which Tree[R, l].cost is less than or equal to $\mathcal{C}$. This solution obtained for this value of l represents the minimum-size subtree that satisfies the constraint $\mathcal{C}$ on the MDL cost (or number of misclassified records). The crucial observation here is that the invocation COMPUTECOST(R, l) can *reuse* the results of the previous invocation COMPUTECOST(R, $l-1$)—thus, costs that were already computed during earlier invocations do not need to be recomputed during the current invocation. Finally, procedure PRUNETOSIZEK is invoked with R and the final (optimal) value of l to prune unnecessary nodes from T.

The space and time complexity of the above algorithm is $O(nl)$, where n is the number of nodes in the complete tree T and l is the number of nodes in the target tree T_f (i.e., the first value of the size bound for which the accuracy constraint $\mathcal{C}$ is satisfied).

5. The integrated approach: Pushing constraints into tree-building

The dynamic programming algorithms presented in Section 4 (as well as those in Almuallim (1996) and Bohanec and Bratko (1994) enforce the user-specified size/accuracy constraints only after a full decision tree has been grown by the building algorithm. As a consequence, substantial effort (both I/O and CPU computation) may be wasted on growing portions of the tree that are subsequently pruned when constraints are enforced. Clearly, by "pushing" size and accuracy constraints into the tree-building phase, significant gains in performance can be attained. In this section, we present such *integrated* decision tree induction algorithms that integrate the constraint-enforcement phase into the tree-building phase instead of performing them one after the other.

Our integrated algorithms are similar to the BUILDTREE procedure depicted in figure 2(a). 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 T_p is pruned using the user-specified size/accuracy constraints.[4] Note, however, the pruning algorithms of Section 4 cannot be used to prune the partial tree.

The problem with applying constraint-based pruning (figure 3) before the full tree has been built is that, in procedure COMPUTECOST, the MDL cost of the cheapest subtree rooted at a leaf N is assumed to be $C(S) + 1$ (Steps 4 and 5). While this is true for the fully-grown tree, 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. Obviously, splitting node N could result in a subtree rooted at N with cost much less than $C(S) + 1$. Thus, $C(S) + 1$ may over-estimate the MDL 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 are actually part of the optimal size- or accuracy-constrained subtree. This is undesirable since the final tree may no longer be the optimal subtree that satisfies the user-specified constraints.

In order to perform constraint-based pruning on a partial tree T_p, and still ensure that only suboptimal nodes are pruned, we adopt an approach that is based on the following observation. (For concreteness, our discussion is based on the case of size constraints.) Suppose U is the cost of the cheapest subtree of size at most k of the partial tree T_p. Note that this subtree may not be the final optimal subtree, since expanding a node in T_p could cause its cost to reduce by a substantial amount, in which case, the node along with its children may be included in the final subtree. U does, however, represent an upper bound on the cost of the final optimal subtree T_f. Now, if we could also compute *lower bounds* on the cost of subtrees of various sizes rooted at nodes of T_p, then we could use these lower bounds to determine the nodes N in T_p such that every potential subtree of size at most k (of the full tree T) containing N is guaranteed to have a cost greater than U. Clearly, such nodes can be safely pruned from T_p, since they cannot possibly be part of the optimal subtree whose cost is definitely less than or equal to U.

While it is relatively straightforward to compute U (using procedure COMPUTECOST on T_p), we still need to (1) estimate the lower bounds on cost at each node of the partial tree T_p, and (2) show how these lower bounds can be combined with the upper bound U (in a "branch-and-bound" fashion) to identify prunable nodes of T_p. We address these issues in the subsections that follow.

5.1. *Computing lower bounds on subtree costs*

To obtain lower bounds on the cost (either MDL cost or number of misclassifications) of a subtree at arbitrary nodes of T_p, we first need to be able to compute lower bounds for subtree costs at leaf nodes that are "yet to be expanded". These bounds can then be propagated "upwards" to obtain lower bounds for other nodes of T_p. Obviously, any subtree rooted at node N must have an MDL cost of at least 1, and thus 1 is a simple, but conservative estimate for the MDL cost of the cheapest subtree at leaf nodes that are "yet to be expanded". In our earlier work (Rastogi and Shim, 1998), we have derived more accurate lower bounds on the MDL cost of subtrees by also considering split costs. More specifically, let S be the set of records at node N and c 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 $1 \leq i < c$ (that is, $n_1, \ldots, n_c$ 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 MDL cost for a subtree at N is $C(S) + 1$. For values of $s > 0$, a lower bound on the MDL cost of encoding a subtree with s splits (or, $2 * s + 1$ nodes) and rooted at node N is given by the following theorem.

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

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 the case that, 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 $c - s - 1$ 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 $c - s - 1$ 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 $c - s - 1$ classes for which $\sum_{i \in L} n_i$ is minimum. Obviously, the above cost is minimum for the $c - s - 1$ classes with the smallest number of records in S, that is, classes $s + 2, \ldots, c$. Thus, the cost for encoding the records in the $s + 1$ leaves of the subtree is at least $\sum_{i=s+2}^{c} n_i$. □

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 MDL-cost subtrees at N with 1 and 2 splits are as shown in figures 4(a) and (b), respectively. The minimum MDL cost for encoding each node is presented next to

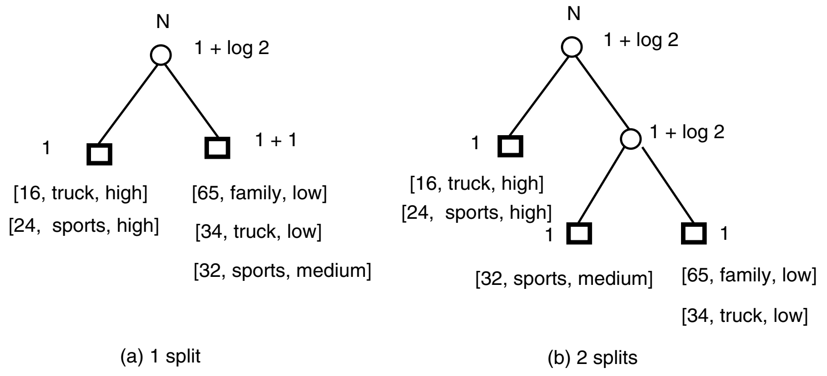


Figure 4. Minimum MDL-cost subtrees with 1 and 2 splits.

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 MDL cost of log 2 for specifying the splitting attribute. Furthermore, in figure 4(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 MDL 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.

The following corollary from the proof of Theorem 5.1 establishes a similar lower bound on the number of misclassified records. (S, c, and $n_1, \ldots, n_c$ are as in Theorem 5.1.)

Corollary 5.3. *The number of misclassified records in any subtree with s splits* ($2 * s + 1$ *nodes*) *and rooted at node N is at least* $\sum_{i=s+2}^{c} n_i$.

Theorem 5.1 and Corollary 5.3 give lower bounds on the MDL cost and number of misclassifications in any subtree with s splits, and can be used to estimate lower bounds on the cost of a subtree with at most l nodes rooted at a node N. Assuming MDL costs and $s = \frac{l-1}{2}$, this is simply the minimum of (1) $C(S) + 1$ (MDL cost of subtree with 0 splits), and (2) for $1 \leq i \leq s$, the lower bound on the MDL cost of a subtree with i splits and rooted at N (as described in Theorem 5.1).

5.2. Computing an optimal size-constrained subtree

Minimum MDL cost. As described earlier, our integrated constraint-pushing strategy involves the following three steps, which we now describe in more detail: (1) compute the cost of the cheapest subtree of size (at most) k of the partial tree T_p (this is an upper bound U on the cost of the final optimal tree T_f); (2) compute lower bounds on the cost of subtrees of varying sizes that are rooted at nodes of the partial tree T_p; and, (3) use the bounds

```
procedure ComputeCostUsingConstraints(Node $N$, integer $l$):
1.  if Tree[$N$, $l$].computed = true
2.      return [Tree[$N$, $l$].realCost, Tree[$N$, $l$].lowCost]
3.  else if $l < 3$ or $N$ is a "pruned" or "not expandable" leaf
4.      Tree[$N$, $l$].realCost := Tree[$N$, $l$].lowCost := $C(S) + 1$
5.  else if $N$ is a "yet to be expanded" leaf {
6.      Tree[$N$, $l$].realCost := $C(S) + 1$
7.      Tree[$N$, $l$].lowCost := lower bound on cost of subtree cost rooted at $N$ with at most $l$ nodes
8.  else {
9.      Tree[$N$, $l$].lowCost := Tree[$N$, $l$].realCost := $C(S) + 1$;
10.     for $k_1$ := 1 to $l - 2$ do {
11.         $k_2 := l - k_1 - 1$
12.         [realCost$_1$, lowCost$_1$] := ComputeCostUsingConstraints($N_1$, $k_1$)
13.         [realCost$_2$, lowCost$_2$] := ComputeCostUsingConstraints($N_2$, $k_2$)
14.         if realCost$_1$ + $C_{split}(N) + 1$ + realCost$_2$ < Tree[$N$, $l$].realCost
15.             Tree[$N$, $l$].realCost := realCost$_1$ + $C_{split}(N) + 1$ + realCost$_2$
16.         if lowCost$_1$ + $C_{split}(N) + 1$ + lowCost$_2$ < Tree[$N$, $l$].lowCost
17.             Tree[$N$, $l$].lowCost := lowCost$_1$ + $C_{split}(N) + 1$ + lowCost$_2$
18.     }
19. }
20. Tree[$N$, $l$].computed := true
21. return [Tree[$N$, $l$].realCost, Tree[$N$, $l$].lowCost]
```

Figure 5. Algorithm for computing minimum MDL-cost subtrees using lower bounds.

computed in steps (1) and (2) to identify and prune nodes that cannot possibly belong to the optimal constrained subtree T_f. Procedure ComputeCostUsingConstraints (depicted in figure 5) accomplishes the first two steps, while procedure PruneUsingConstraints (depicted in figure 6) achieves step (3).

Procedure ComputeCostUsingConstraints distinguishes among three classes of leaf nodes in the partial tree. The first class includes leaf nodes that still need to be expanded ("yet to be expanded"). The two other classes consist of leaf nodes that are either the result of a pruning operation ("pruned") or cannot be expanded any further because they are pure ("not expandable"). ComputeCostUsingConstraints uses dynamic programming to compute in Tree[N, l].realCost the MDL cost of the cheapest subtree of size at most l that is rooted at N in the partially-built tree—this is similar to procedure ComputeCost (figure 3). In addition, ComputeCostUsingConstraints also computes in Tree[N, l].lowCost, a lower bound on the MDL cost of the cheapest subtree with size at most l that is rooted at N (if the partial tree were expanded fully)—the lower bounds on the MDL cost of subtrees rooted at "yet to be expanded" leaf nodes (Theorem 5.1) are used for this purpose. The only difference between the computation of the real costs and the lower bounds is that, for a "yet to be expanded" leaf node N, the former uses $C(S) + 1$ while the latter uses the lower bound for the minimum MDL-cost subtree rooted at N. Procedure ComputeCostUsingConstraints is invoked with input parameters R and k, where R is the root of T_p and k

```
procedure PRUNEUSINGCONSTRAINTS(Node $N$, integer $l$, real $B$):
1.  Mark node $N$
2.  if $B \leq$ Bound[$N$, $l$] return
3.  for $i := 1$ to $l$ do
4.      if $B >$ Bound[$N$, $i$]
5.          Bound[$N$, $i$] := $B$
6.  if Tree[$N$, $l$].lowCost $> B$ or Tree[$N$, $l$].lowCost $= C(S) + 1$ return
7.  else if $N$ is not a leaf node and $l \geq 3$ {
8.      for $k_1 := 1$ to $l - 2$ do {
9.          $k_2 := l - k_1 - 1$
10.         if $C_{split}(N) + 1+$ Tree[$N_1$, $k_1$].lowCost + Tree[$N_2$, $k_2$].lowCost $\leq B$ {
11.             $B_1 := B - (C_{split}(N) + 1)-$ Tree[$N_2$, $k_2$].lowCost
12.             $B_2 := B - (C_{split}(N) + 1)-$ Tree[$N_1$, $k_1$].lowCost
13.             PRUNEUSINGCONSTRAINTS($N_1$, $k_1$, $B_1$);
14.             PRUNEUSINGCONSTRAINTS($N_2$, $k_2$, $B_2$);
15.         }
16.     }
17. }
```

Figure 6. Branch-and-bound pruning algorithm.

is the constraint on the number of nodes. Again, note that U = Tree[R, k].realCost represents an upper bound on the cost of the final optimal subtree satisfying the user-specified constraints.

Once the real costs and lower bounds are computed, the next step is to identify *prunable* nodes N in T_p and prune them. A node N in T_p is prunable if every potential subtree of size at most k (after "yet to be expanded leaves" in T_p are expanded) that contains node N is guaranteed to have an MDL cost greater than Tree[R, k].realCost. Invoking procedure PRUNEUSINGCONSTRAINTS (illustrated in figure 6) with input parameters R (root node of T_p), k, and Tree[R, k].realCost (upper bound on the cost of T_f) ensures that *every non-prunable* node in T_p is *marked* (see Theorem 5.4, below). Thus, after PRUNEUSING-CONSTRAINTS completes execution, it is safe to prune all unmarked nodes from T_p, since these cannot possibly be in the MDL-optimal subtree T_f with size at most k.

Intuitively, procedure PRUNEUSINGCONSTRAINTS works by using the computed lower bounds at nodes of T_p in order to "propagate" the upper bound (Tree[R, k].realCost) on the cost of T_f down the partial tree T_p (Steps 11–14). Assume that some node N (with children N_1 and N_2) is reached with a "size budget" of l and a cost bound of B. If there exists some distribution of l among N_1 and N_2 such that the sum of the corresponding lower bounds does not exceed B (Steps 8–10), then N_1 and N_2 may belong the optimal subtree and PRUNEUSINGCONSTRAINTS is invoked recursively (Steps 11–14) to (a) mark N_1 and N_2 (Step 1), and (b) search for nodes that need to be marked in the corresponding subtrees. Thus, nodes N_1 and N_2 will be left unmarked if and only if, for every possible size budget that reached N, no combination was ever found that could beat the corresponding upper bound B.

More formally, consider a node N' in the subtree of T_p rooted at N_1 and let l and B denote the size budget and cost upper bound propagated down to N (parent of N_1 and N_2). We say that N' is *prunable with respect to* (N, l, B) if every potential subtree of size at most l (after T_p is fully expanded) that is rooted at N and contains N', has an MDL cost greater than B. PRUNEUSINGCONSTRAINTS is based on the following key observation: If N' is *not prunable* with respect to (N, l, B), then, for some $1 \leq k_1 \leq l - 2$,

1. $C_{split}(N) + 1 +$ Tree$[N_1, k_1]$.lowCost $+$ Tree$[N_2, l - k_1 - 1]$.lowCost $\leq B$, and
2. N' is not prunable with respect to $(N_1, k_1, B - (C_{split}(N) + 1) -$ Tree$[N_2, l - k_1 - 1]$.lowCost$)$.

That is, if N' is not prunable with respect to (N, l, B) then there exists a way to distribute the size budget l along the path from N down to N' such that the lower bounds on the MDL cost never exceed the corresponding upper bounds, on all the nodes in the path. Obviously, N' is not prunable (i.e., should be marked) if it is not prunable with respect to *some* triple (N, l, B). Based on these observations, we can formally prove the correctness of procedure PRUNEUSINGCONSTRAINTS.

Theorem 5.4. *If a node in T_p is not prunable, then it is marked by procedure* PRUNEUSINGCONSTRAINTS.

Proof: Let N be a node in T_p that is not prunable. We need to show that it is marked by the procedure. Since N is not prunable, there must exist a subtree of size at most k containing N (after "yet to be expanded" leaves in the partial tree have been expanded) whose cost is less than or equal to Tree$[R, k]$.realCost—let T' be this subtree. Let $N_1 = R, \ldots, N_m = N$ be the sequence of nodes from R to N in T'. We use induction to show that each node N_i along the path is marked by PRUNEUSINGCONSTRAINTS. More specifically, we show that PRUNEUSINGCONSTRAINTS is invoked for each node N_i along the path with l greater than or equal to the number of nodes in the subtree rooted at N_i in T' and B greater than or equal to the cost of the subtree rooted at N_i in T'.

Clearly, the base case holds since $N_1 = R$ is marked initially when procedure PRUNEUSINGCONSTRAINTS is first invoked on R, and T' contains at most k nodes and cost of T' is at most Tree$[R, k]$.realCost. Suppose the condition holds for N_i. We show that it also holds for N_{i+1}. Let S_1 and S_2 denote the left and right subtrees of N_i in T' rooted at its left and right children, N_1 and N_2, respectively. Without loss of generality, let $N_1 = N_{i+1}$. Let k_i and C_i denote the number of nodes and the cost of subtree S_i, respectively. We need to show that PRUNEUSINGCONSTRAINTS is invoked for N_{i+1} with $l \geq k_1$ and $B \geq C_1$ (this also ensures that N_{i+1} gets marked). Consider the invocation of PRUNEUSINGCONSTRAINTS with parameters N_i, l and B. By the induction hypothesis, $B \geq C_{split}(N_i) + 1 + C_1 + C_2$. From the definition of lower bounds, $C_1 \geq$ Tree$[N_1, k_1]$.lowCost and $C_2 \geq$ Tree$[N_2, k_2]$.lowCost, and thus it follows that $B \geq$ Tree$[N_i, l]$.lowCost. Also, since T' is compact, Tree$[N_i, l]$.lowCost $\leq C_{split}(N_i) + 1 + C_1 + C_2 < C(S) + 1$. Thus, since N_i is not a leaf node $l \geq 3$, and $C_{split}(N_i) + 1 + C_1 + C_2 \leq B$, it follows that procedure PRUNEUSINGCONSTRAINTS is invoked with N_{i+1}, k_1 and $B_1 \geq C_1$ (Step 9). □

As an optimization, procedure PRUNEUSINGCONSTRAINTS maintains the array Bound[] in order to reduce computational overheads. Each entry Bound[N, l] is initialized to 0 and is used to keep track of the maximum value of B with which PRUNEUSINGCONSTRAINTS has been invoked on node N with size budget $l' \geq l$. The key observation here is that if a node N' in the subtree rooted at N is not prunable with respect to (N, l, B), then it is also not prunable with respect to (N, l', B'), for all $B' \geq B, l' \geq l$. Intuitively, this says that if we have already reached node N with a cost bound B' and size budget l', then invoking PRUNEUSINGCONSTRAINTS on N with a smaller bound $B \leq B'$ and smaller size budget $l \leq l'$ cannot cause any more nodes under N to be marked. Thus, when such a situation is detected, our marking procedure can simply return (Step 2).

Note that when unmarked nodes are pruned from T_p, they are also deleted from the queue Q maintained in the BUILDTREE procedure—this ensures that they are not expanded during the building phase. Further, at the end of the building phase, we still need to run procedures COMPUTECOST and PRUNETOSIZEK (Section 4) to derive the final optimal subtree T_f that satisfies the user-specified size constraint. This is because procedure PRUNEUSINGCONSTRAINTS marks nodes conservatively, which implies that the tree returned by PRUNEUSINGCONSTRAINTS may contain multiple subtrees with minimal MDL cost. This final step is then necessary to select one subtree to return to the user.

Minimum number of misclassified records. Our integrated algorithms for optimizing MDL cost can readily be modified to compute the optimal size-constrained subtree which minimizes the number of misclassified records. The basic idea is to (1) set the real cost for every leaf node equal to the number of misclassified records in the leaf, and (2) compute the lower bounds on subtree costs using Corollary 5.3.

Time and space complexity. The time complexity of Procedure COMPUTECOSTUSINGCONSTRAINTS is $O(nk)$, where n is the number of nodes in T_p and k is the upper bound on the desired tree size. The worst-case time complexity of procedure PRUNEUSINGCONSTRAINTS is exponential in k—however, in practice, as indicated by our experiments, it is much lower (close to linear in k) due to our optimizations involving the Bound[] array. The space complexity is $O(nk)$ (to store the Tree[] and Bound[] arrays).

5.3. *Computing an optimal accuracy-constrained subtree*

We now discuss how procedures COMPUTECOSTUSINGCONSTRAINTS and PRUNEUSINGCONSTRAINTS can be used to compute the subtree with the minimum number of nodes and a certain maximum user-specified cost $\mathcal{C}$. The key idea is to first compute, for the partial tree T_p, the smallest l for which there exists a subtree of T_p whose cost does not exceed the user-specified cost constraint $\mathcal{C}$. This can be computed (as described in Section 4) by repeatedly invoking procedure COMPUTECOSTUSINGCONSTRAINTS with increasing values of l until an l is reached at which Tree[R, l].realCost falls below or equals $\mathcal{C}$. Note that, concurrently with the real cost of subtrees, the procedure also computes lower bounds on subtree costs for nodes in T_p. Thus, PRUNEUSINGCONSTRAINTS can be invoked on the root R of T_p with inputs (1) the final value of l, i.e., the value at which the cost of a subtree (of size l) of T_p falls below $\mathcal{C}$, and (2) the user-specified constraint $\mathcal{C}$. This results in nodes for whom there could (potentially) exist a subtree of size at most l with cost less than or equal to $\mathcal{C}$, being marked. Unmarked nodes can then be pruned from T_p.

6. Experimental results

In order to investigate the performance gains that can be realized as a result of *pushing* constraints into the building phase, we conducted experiments on real-life as well as synthetic data sets. We used the PUBLIC (Rastogi and Shim, 1998) algorithm to construct decision trees. The PUBLIC algorithm integrates the building and pruning phases that were described separately in Section 3. However, it does not perform any constraint-based pruning. Thus, we extended the PUBLIC algorithm to (1) enforce size/accuracy constraints *after* the build phase (as described in Section 4), and (2) push size/accuracy constraints *during* the build phase (as described in Section 5). We refer to the former algorithm as PUBLIC *without* constraint pushing, while the latter algorithm is referred to as PUBLIC *with* constraint pushing. We only considered the size constraint with MDL cost in our experiments—thus, for a given k, we were interested in computing the tree with at most k nodes and the minimum MDL cost.

Since real-life data sets are generally small, we also used synthetic data sets to study the benefits of constraint pushing on larger data collections. The primary purpose of the synthetic data sets was to examine sensitivity to parameters such as noise, number of classes, and number of attributes. Synthetic data sets allowed us to vary the above parameters in a controlled fashion. All of our experiments were performed using a Sun Ultra-60 machine with 512 MB of RAM and running Solaris 2.7.

Our experimental results with both real-life and synthetic data sets clearly demonstrate the effectiveness of integrating user-specified constraints into the tree-building phase. We found that our constraint-pushing algorithms always result in significant reductions in execution times that are sometimes as high as two or three orders of magnitude.

6.1. Algorithms

In our experiments, we compared the execution times and the number of nodes generated for four algorithms, whose characteristics we summarize below.

- *PUBLIC(1) with/without constraint pushing*: This is PUBLIC with/without constraint pushing with the very conservative estimate of 1 as the cost of the cheapest subtree rooted at a "yet to be expanded" leaf node.
- *PUBLIC(S) with/without constraint pushing*: This is PUBLIC with/without constraint pushing; it considers subtrees with splits for the minimum cost subtree at a "yet to be expanded" leaf node, and includes the cost of specifying the splitting attribute for splits (see Theorem 5.1).

The constraint-pushing algorithms are implemented using the same code base as PUBLIC except that they perform size-constraint-based pruning while the tree is being built. The

Table 2. Real-life data sets.

Data set	No. of attributes	No. of classes	No. of records	Nodes in final tree
Letter	16	26	13368	1989
Satimage	36	7	4435	185

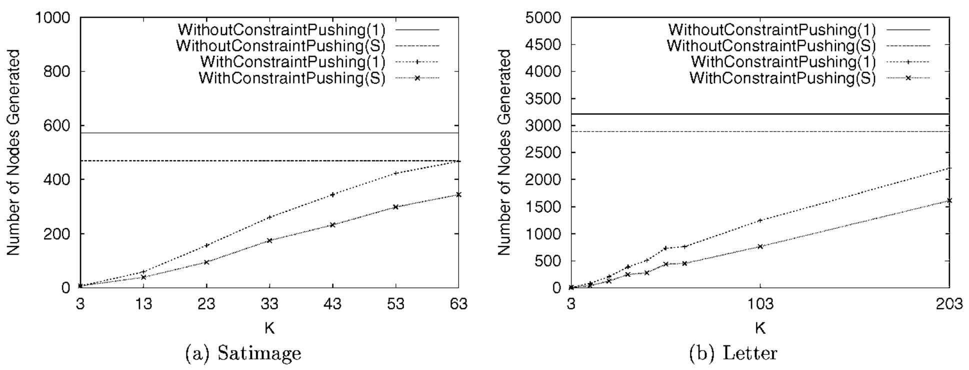
tree itself is built breadth-first, and the pruning procedure is invoked repeatedly for each level, after all the nodes at the level have been split.

6.2. *Real-life data sets*

We experimented with two real-life data sets whose characteristics are illustrated in Table 2. These data sets were obtained from the UCI Machine Learning Repository.[5] Data sets in the UCI Machine Learning Repository often do not have both training and test data sets. For these data sets, we randomly chose 2/3 of the data and used it as the training data set. The last column in Table 2 contains the number of nodes in the final tree constructed by PUBLIC without constraint pruning.

6.3. *Results with real-life data sets*

For the two real-life data sets, we plot the number of nodes generated by the algorithms and their execution times in figures 7 and 8, respectively. In our experiments, we vary k, the constraint on the number of nodes in the final tree. 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. From figure 7, it follows that the constraint-pushing algorithms generate significantly fewer nodes than the algorithms that enforce the size constraint only after the build is complete. The reductions are much larger for smaller values of k and, in a number of cases, exceed two orders of magnitude. The improvements in the number of nodes split are also reflected in the graphs for execution



(a) Satimage (b) Letter

Figure 7. Real-life data sets: Number of nodes generated.

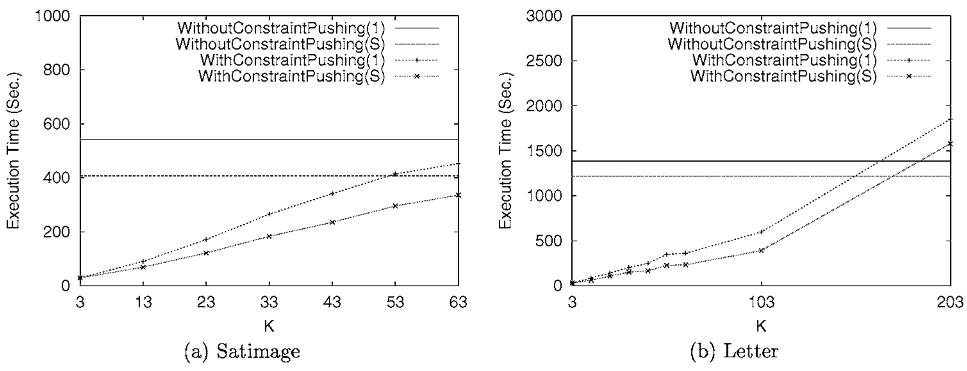


Figure 8. Real-life data sets: Execution time (secs).

times in figure 8. The only exception is when $k = 203$ for the Letter data set. Here, the computational overhead of the branch-and-bound pruning algorithm causes the running times of the constraint-pushing algorithms to be higher, even though they generate fewer nodes. However, since trees with more than 100 nodes are difficult to assimilate and interpret, we typically expect the value of k to be fairly small (less than 200) in most cases. Thus, from the graphs, we can conclude that, in general, pushing tree-size constraints into the building phase does indeed yield substantial performance speedups.

Note that the number of nodes generated by algorithms that enforce the size constraint only after the building phase has completed, is a constant, independent of k. In contrast, the integrated constraint-pushing algorithms are sensitive to k. As k is increased, the final optimal tree contains more nodes and fewer nodes are pruned by our branch-and-bound pruning algorithm. We should point out, however, that the pruning is still effective and the number of nodes generated increases linearly with k.

6.4. Synthetic data sets

In order to study the sensitivity of our algorithms 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), Rastogi and Shim (1998), and Shafer et al. (1996) and available from the IBM Quest home page.[6] 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 depicted in Table 3. Among the attributes, elevel, car, and zipcode are categorical, while all others are numeric. Different data distributions were generated by using one of ten distinct classification functions to assign class labels to records. We only considered functions 3, 4, 5, and 6 for our experiments, since we found these to be a representative set. Function 3 uses predicates over two attributes, while functions 4, 5, and 6 have predicates with ranges on three attributes. Further details on these functions 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

Table 3. 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

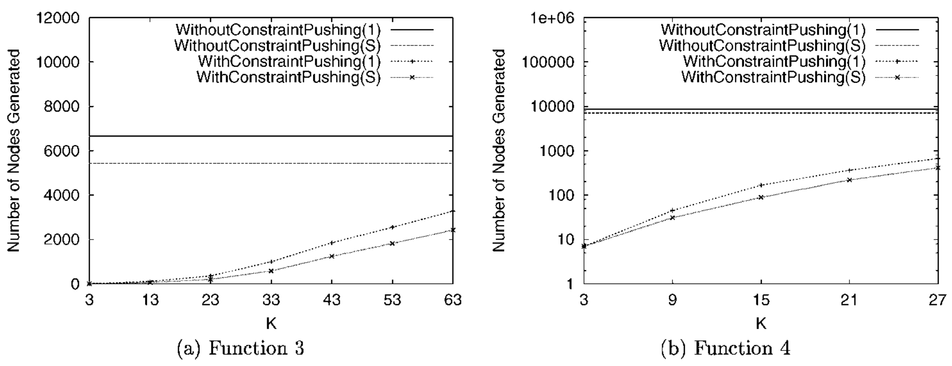


Figure 9. Synthetic data sets: Number of nodes generated.

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 was set to 50000.

6.5. *Results with synthetic data sets*

The results for synthetic data sets are similar to those for real-life data sets, and are illustrated for Functions 3 and 4 in figures 9 and 10. For each data set, the maximum value that we consider for k is about half of the number of nodes in the final tree constructed by PUBLIC without any constraints. These numbers, for the four functions, are presented in Table 4. For each data set, the noise factor was set to 10%. From the figures, it is easy to see that the constraint-pushing algorithms outperform the others by a significant margin. For smaller values of k, performance speedups of more than an order of magnitude are easily realized as a result of pushing size constraints. While the performance gains become smaller as k is

Table 4. Synthetic data sets: Number of nodes without a constraint.

Function No.	3	4	5	6
Nodes in final tree	119	63	217	175

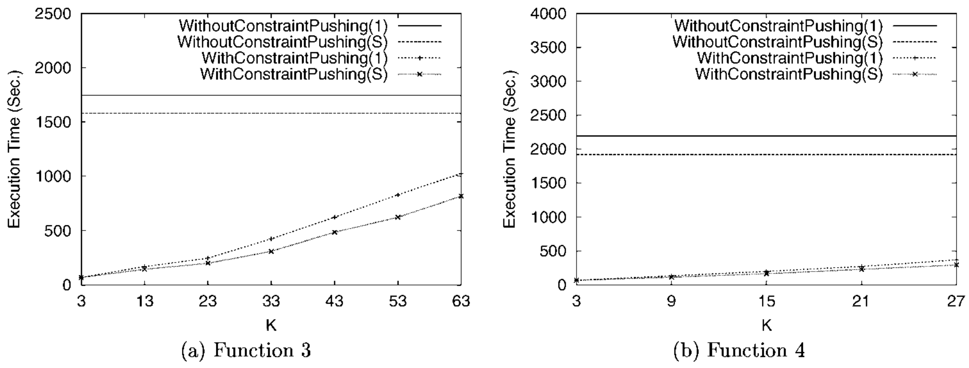


Figure 10. Synthetic data sets: Execution time (secs).

increased, they still remain significant, ranging from 100% for Function 3 when $k = 63$ to 1000% for Function 4 when $k = 27$.

We also performed experiments to study the effects of noise on the performance of our algorithms. We varied noise from 2% to 10% for every function, and found that the execution times of the algorithms on all the data sets were very similar. As a result, in figures 11 and 12, we only plot the number of generated nodes and execution times for Functions 5 and 6. We fixed the size constraint k to be 33 for both data sets. From the graphs, it follows that both execution times and the number of nodes generated 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 the algorithms without constraint

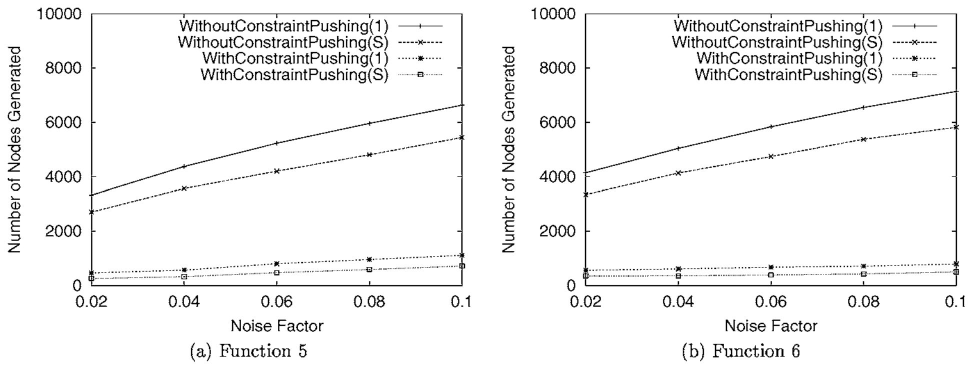


Figure 11. Synthetic data sets: Number of nodes generated.

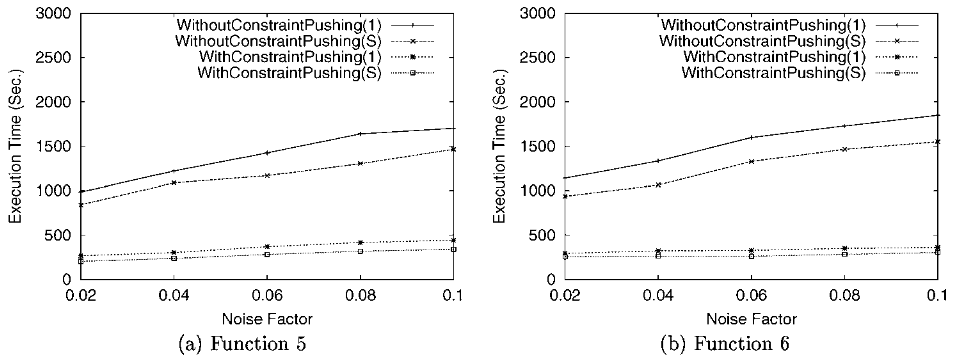


Figure 12. Synthetic data sets: Execution time (secs).

pushing increase at a faster rate than those for the integrated algorithms as the noise factor is increased. Thus, integrating size constraints with tree building results in better performance improvements at higher noise values.

7. Conclusions

In this paper, we have proposed a general framework that enables users to specify constraints on the size and accuracy of decision trees. The motivation for such constraints is to allow the efficient construction of decision tree classifiers that are easy to interpret and, at the same time, have good accuracy properties.

We have proposed novel algorithms for pushing size and accuracy constraints into the tree-building phase. Our algorithms use a combination of dynamic programming and branch-and-bound techniques to prune early (during the growing phase) portions of the partially-built tree that cannot possibly be part of the optimal subtree that satisfies the user-specified constraints. Enforcing the constraints while the tree is being built prevents a significant amount of effort being expended on expanding nodes that are not part of the optimal subtree. Our experimental results with real-life and synthetic data sets corroborate this fact, and clearly demonstrate the effectiveness of our integrated constraint-enforcement and building algorithms. Our proposed integrated algorithms deliver significant performance speedups that are, in many cases, in the range of two or three orders of magnitude.

Acknowledgments

The work of Dongjoon Hyun and Kyuseok Shim was partially supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

Notes

1. To simplify the presentation, we concentrate on binary decision trees in the remainder of the paper. However, our algorithms can be extended to handle the more general case of k-ary splits in a straightforward manner.

2. All logarithms in the paper are to the base 2.
3. If there are multiple subtrees of T satisfying the constraint, then our algorithms compute one of them.
4. Determining a "good" value for the frequency of the pruning operation depends on a number of different factors, including data-set and memory sizes and the actual tradeoff between pruning cost and benefit. In practice, we have found that the simple heuristic rule of invoking the pruning procedure *once per level of the tree* (i.e., after all nodes at a given level have been split) performs reasonably well (Section 6).
5. Available at `http://www.ics.uci.edu/~mlearn/MLRepository.html`.
6. The URL for the page is `http://www.almaden.ibm.com/cs/quest/demos.html`.

References

Agrawal, R., Ghosh, S.P., Imielinski, T., Iyer, B.R., and Swami, A.N. 1992. An interval classifier for database mining applications. In Proceedings of the 18th International Conference on Very Large Data Bases, Vancouver, Canada, 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.

Almuallim, H. 1996. An efficient algorithm for optimal pruning of decision trees. Artificial Intelligence, 83:346–362.

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

Bohanec, M. and Bratko, I. 1994. Trading accuracy for simplicity in decision trees. Machine Learning, 15:223–250.

Breiman, L., Friedman, J.H., Olshen, R.A., and Stone, C.J. 1984. Classification and Regression Trees. Chapman and Hall.

Cheeseman, P., Kelly, J., Self, M. et al. 1988. AutoClass: A Bayesian classification system. In 5th Int'l Conf. on Machine Learning. 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 Proceedings of the 22nd International Conference on Very Large Data Bases, Bombay, India.

Gehrke, J., Ganti, V., Ramakrishnan, R., and Loh, W.-Y. 1999. BOAT—optimistic decision tree construction. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, Philadelphia, Pennsylvania.

Gehrke, J., Ramakrishnan, R., and Ganti, V. 1998. RainForest—A framework for fast decision tree construction of large datasets. In Proceedings of the 24th International Conference on Very Large Data Bases, New York, USA.

Goldberg, D.E. 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Morgan Kaufmann.

Hunt, E.B., Marin, J., and Stone, P.J. (Eds.). 1966. Experiments in Induction. Academic Press, New York.

Krichevsky, R. and Trofimov, V. 1981. The performance of universal encoding. IEEE Transactions on Information Theory, 27(2):199–207.

Mehta, M., Agrawal, R., and Rissanen, J. 1996. SLIQ: A fast scalable classifier for data mining. In Proceedings of the Fifth International Conference on Extending Database Technology (EDBT'96), Avignon, France.

Mehta, M., Rissanen, J., and Agrawal, R. 1995. MDL-based decision tree pruning. In Proceedings of the First International Conference on Knowledge Discovery and Data Mining, Montreal, Canada.

Mitchie, D., Spiegelhalter, D.J., and Taylor, C.C. 1994. Machine Learning, Neural and Statistical Classification. Ellis Horwood.

Murthy, S.K. 1998. Automatic construction of decision trees from data: A multi-disciplinary survey. Data Mining and Knowledge Discovery, 2(4):345–389.

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. and Rivest, R.L. 1989. Inferring decision trees using minimum description length principle. Information and Computation, 80(3):227–248.

Quinlan, J.R. 1993. C4.5: Programs for Machine Learning. Morgan Kaufman.

Rastogi, R. and Shim, K. 1998. PUBLIC: A decision tree classifier that integrates building and pruning. In Proceedings of the 24th International Conference on Very Large Data Bases, New York, USA, pp. 404–415.

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 Proceedings of the 22nd International Conference on Very Large Data Bases, Mumbai (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 Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, Newport Beach, California.