Efficient Algorithms for Constructing Decision Trees with Constraints

Minos Garofalakis
Bell Laboratories
minos@bell-labs.com

Dongjoon Hyun
KAIST* and AITrc†
djhyun@cs.kaist.ac.kr

Rajeev Rastogi
Bell Laboratories
rastogi@bell-labs.com

Kyuseok Shim
KAIST* and AITrc†
shim@cs.kaist.ac.kr

ABSTRACT

Classification is an important problem in data mining. 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.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data mining*

General Terms

Classification, Decision tree

1. INTRODUCTION

Classification is an important problem in data mining. Briefly, the input to a classifier is a *training set* of records, each of which is a tuple 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. Among the different proposals, decision tree classifiers have found the widest applicability in large-scale data mining environments. A number of algorithms for inducing decision trees have been proposed over the years (e.g., C4.5 [8], CART [3], SPRINT [11], PUBLIC [9], BOAT [5]). Most of these algorithms consist of two distinct phases, a *building* (or *growing*) phase followed by a *pruning* phase.

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. 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; thus, they 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 [2] 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!

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 [6, 7,

*Korea Advanced Institute of Science and Technology
†Advanced Information Technology Research Center

KDD 2000, Boston, MA USA

9] 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. Due to space constraints, we defer the presentation of our algorithms for constructing accuracy-constrained decision trees to [4].

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. Such algorithms for the optimal pruning of accurate decision trees have been proposed in the earlier work of Bohanec and Bratko [2] and Almuallim [1], where the accuracy measure was assumed to be the number of misclassified training set records (i.e., the "resubstitution error" of [3]).

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 them 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 [9] 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 (see [4] for details). The performance speedups, in some cases, can be as high as two or three orders of magnitude.

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

Table 1: Notation

2. 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 algorithms can be found in [3, 9, 11]. Table 1 summarizes some of the notation used throughout this paper.

The overall algorithm for growing a decision tree classifier is depicted in Figure 1(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 each internal node of the tree is selected so that it minimizes an *impurity function*, such as the *entropy*, of the induced data partitioning [3].

```
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 N1 and N2
9.        Append N1 and N2 to Q
10.    }
11. }
```

(a)

```
procedure PruneTree(Node N):
1.  if N is a leaf return (C(S) + 1)
2.  minCost1 := PruneTree(N1);
3.  minCost2 := PruneTree(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
```

(b)

Figure 1: (a) Tree-Building Algorithm (b) Tree-Pruning Algorithm

To prevent overfitting of the training data, the MDL principle [10] is applied to prune the tree built in the growing phase and make it more general. Briefly, the MDL principle states that the "best" tree is the one that can be encoded using the smallest number of bits. The cost of encoding the tree comprises three distinct components: (1) the cost of encoding the structure of the tree (single bit); (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. In the rest of the paper, we refer to the cost of encoding a tree computed above as the *MDL cost* of the tree.

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 1(b) gives the pseudocode for the pruning procedure; more details can be found in [9].

3. THE INTEGRATED APPROACH: PUSHING CONSTRAINTS INTO TREE-BUILDING

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 the following constraint on the desired subtree T_f of T: For a given k, T_f contains at most k nodes and has the minimum possible MDL cost.

The dynamic programming algorithms presented in [1, 2] 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 1(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. Note, however, the pruning algorithm of Section 2 cannot be used to prune the partial tree.

The problem with applying constraint-based pruning before the full tree has been built is that, in procedure PruneTree (Figure 1(b)), the MDL cost of the cheapest subtree rooted at a leaf N is assumed to be $C(S) + 1$ (Step 1). 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 overestimate 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

```
procedure ComputeCostUsingConst(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 k1 := 1 to l − 2 do {
11.       k2 := l − k1 − 1
12.       [realCost1, lowCost1] :=
                     ComputeCostUsingConst(N1, k1)
13.       [realCost2, lowCost2] :=
                     ComputeCostUsingConst(N2, k2)
14.       if realCost1 + Csplit(N) + 1 + realCost2 <
                        Tree[N, l].realCost
15.          Tree[N, l].realCost := realCost1 + Csplit(N)+
                        1 + realCost2
16.       if lowCost1 + Csplit(N) + 1 + lowCost2 <
                        Tree[N, l].lowCost
17.          Tree[N, l].lowCost := lowCost1 + Csplit(N)+
                        1 + lowCost2
18.    }
19. }
20. Tree[N, l].computed := true
21. return [Tree[N, l].realCost, Tree[N, l].lowCost]
```

Figure 2: Algorithm for Computing Minimum MDL-Cost Subtrees using Lower Bounds

or equal to U.

While it is relatively straightforward to compute U, 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.

3.1 Computing Lower Bounds on Subtree Costs

To obtain lower bounds on the MDL cost 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 [9], we have derived more accurate lower bounds on the MDL cost of subtrees by also considering split costs.

3.2 Computing an Optimal Size-Constrained Subtree

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

```
procedure PruneUsingConst(Node N, integer l, real B):
1.  Mark node N
2.  if B ≤ 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
7.     return
8.  else if N is not a leaf node and l ≥ 3 {
9.     for k1 := 1 to l − 2 do {
10.       k2 := l − k1 − 1
11.       if Csplit(N) + 1+ Tree[N1, k1].lowCost +
                        Tree[N2, k2].lowCost ≤ B {
12.          B1 := B − (Csplit(N) + 1)− Tree[N2, k2].lowCost
13.          B2 := B − (Csplit(N) + 1)− Tree[N1, k1].lowCost
14.          PruneUsingConst(N1, k1, B1);
15.          PruneUsingConst(N2, k2, B2);
16.       }
17.    }
18. }
```

Figure 3: Branch-and-Bound Pruning Algorithm

Procedure ComputeCostUsingConst 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"). ComputeCostUsingConst 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. In addition, ComputeCostUsingConst 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 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 ComputeCostUsingConst is invoked with input parameters R and k, where R is the root of T_p and k 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 PruneUsingConst(illustrated in Figure 3) 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*. Thus, after PRUNEUSINGCONST 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 PRUNEUSINGCONST 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 12–15). 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 9–11), then N_1 and N_2 may belong the optimal subtree and PRUNEUSINGCONST is invoked recursively (Steps 12–15) 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. PRUNEUSINGCONST 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 that if a node in T_p is not prunable, then it is marked by procedure PRUNEUSINGCONST. (The proof can be found in [4].)

As an optimization, procedure PRUNEUSINGCONST 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 PRUNEUSINGCONST 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 PRUNEUSINGCONST 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).

4. 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 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).

5. REFERENCES

[1] Hussein Almuallim. "An efficient algorithm for optimal pruning of decision trees". *Artificial Intelligence*, 83:346–362, 1996.

[2] Marko Bohanec and Ivan Bratko. "Trading Accuracy for Simplicity in Decision Trees". *Machine Learning*, 15:223–250, 1994.

[3] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *"Classification and Regression Trees"*. Chapman & Hall, 1984.

[4] Minos Garofalakis, Dongjoon Hyun, Rajeev Rastogi, and Kyuseok Shim. "Efficient Algorithms for Constructing Decision Trees with Constraints". March 2000. Bell Laboratories Tech. Memorandum.

[5] Johannes Gehrke, Venkatesh Ganti, Raghu Ramakrishnan, and Wei-Yin Loh. "BOAT – Optimistic Decision Tree Construction". In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, Pennsylvania, May 1999.

[6] Manish Mehta, Jorma Rissanen, and Rakesh Agrawal. MDL-based decision tree pruning. In *Int'l Conference on Knowledge Discovery in Databases and Data Mining (KDD-95)*, Montreal, Canada, August 1995.

[7] J. R. Quinlan and R. L. Rivest. Inferring decision trees using minimum description length principle. *Information and Computation*, 1989.

[8] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1993.

[9] Rajeev Rastogi and Kyuseok Shim. "PUBLIC: A Decision Tree Classifier that Integrates Building and Pruning". In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 404–415, New York, USA, August 1998.

[10] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[11] John Shafer, Rakesh Agrawal, and Manish Mehta. "SPRINT: A Scalable Parallel Classifier for Data Mining". In *Proceedings of the 22nd International Conference on Very Large Data Bases*, Mumbai(Bombay), India, September 1996.