Data Mining and Knowledge Discovery, 4, 127–162, 2000

RainForest—A Framework for Fast Decision Tree Construction of Large Datasets

JOHANNES GEHRKE
RAGHU RAMAKRISHNAN
VENKATESH GANTI
Department of Computer Sciences, University of Wisconsin-Madison

Editor: Chaudhuri

Abstract. Classification of large datasets is an important data mining problem. Many classification algorithms have been proposed in the literature, but studies have shown that so far no algorithm uniformly outperforms all other algorithms in terms of quality. In this paper, we present a unifying framework called RainForest for classification tree construction that separates the *scalability* aspects of algorithms for constructing a tree from the central features that determine the *quality* of the tree. The generic algorithm is easy to instantiate with specific split selection methods from the literature (including C4.5, CART, CHAID, FACT, ID3 and extensions, SLIQ, SPRINT and QUEST).

In addition to its generality, in that it yields scalable versions of a wide range of classification algorithms, our approach also offers performance improvements of over a factor of three over the SPRINT algorithm, the fastest scalable classification algorithm proposed previously. In contrast to SPRINT, however, our generic algorithm requires a certain minimum amount of main memory, proportional to the *set* of distinct values in a column of the input relation. Given current main memory costs, this requirement is readily met in most if not all workloads.

Keywords: data mining, decision trees, classification, scalability

1. Introduction

Classification and regression are important data mining problems (Fayyad et al., 1996). The input is a dataset of *training records* (also called *training database*), where each record has several attributes. Attributes whose domain is numerical are called *numerical attributes*, whereas attributes whose domain is not numerical are called *categorical attributes*.[1] There is one distinguished attribute called the *dependent attribute*. The remaining attributes are called *predictor attributes*; they are either numerical or categorical in nature. If the dependent attribute is categorical, the problem is referred to as a *classification problem* and we call the dependent attribute the *class label*. (We will denote the elements of the domain of the class label attribute as *class labels*; the semantics of the term class label will be clear from the context.) If the dependent attribute is numerical, the problem is referred to as a *regression problem*. The goal of classification and regression is to build a concise model of the distribution of the dependent attribute in terms of the predictor attributes. The resulting model is used to assign values to a database of *testing records* where the values of the predictor attributes are known but the value of the dependent attribute is unknown. Classification and

regression have a wide range of applications, including scientific experiments, medical diagnosis, fraud detection, credit approval, and target marketing (Brachman et al., 1996; Inman, 1996; Fayyad et al., 1996). In this paper we concentrate mainly on classification problems.

Many classification models have been proposed in the literature: Neural networks (Sarle, 1994; Kohonen, 1995; Bishop, 1995; Ripley, 1996), genetic algorithms (Goldberg, 1989), Bayesian methods (Cheeseman et al., 1988; Cheeseman and Stutz, 1996), log-linear models and other statistical methods (James, 1985; Agresti, 1990; Chirstensen, 1997), decision tables (Kohavi, 1995), and tree-structured models, so-called *classification trees* (Sonquist et al., 1971; Gillo, 1972; Morgan and Messenger, 1973; Breiman et al., 1984). (There exist excellent overviews of classification methods (Weiss and Kulikowski, 1991; Michie et al., 1994a; Hand, 1997).)

Classification trees are especially attractive in a data mining environment for several reasons. First, due to their intuitive representation, the resulting classification model is easy to assimilate by humans (Breiman et al., 1984; Mehta et al., 1996). Second, classification trees are non-parametric. Classification tree construction algorithms do not make any assumptions about the underlying distribution and are thus especially suited for exploratory knowledge discovery. Third, classification trees can be constructed relatively fast compared to other methods (Mehta et al., 1996; Shafer et al., 1996; Lim et al., 1997). Last, the accuracy of classification trees is comparable or superior to other classification models (Murthy, 1995; Lim et al., 1997; Hand, 1997). In this paper, we restrict our attention to classification trees.[2]

There exists a large number of algorithms to construct classification trees. Most algorithms in the machine learning and statistics community are main memory algorithms, even though today's databases are in general much larger than main memory (Agrawal, 1993). There have been several approaches to dealing with large databases. One approach is to apply a discretization function f to each record in the training database D and run an in-memory algorithm on the discretized database D_f (Quinlan, 1986; Fayyad and Irani, 1993; Quinlan, 1993). Discretization can reduce the main memory requirements as follows: Assume that the discretization function f maps a set of records $S = \{t_1, \ldots, t_n\}$ to the same image record t', i.e. $f(t_i) = t'$ for all $i \in \{1, \ldots, n\}$. Then in the discretized training database D_f, instead of storing the set S, it is sufficient to store the pair $[t', m]$, indicating that record t' appears m times in D_f. This approach has two caveats: First, if the number of predictor attributes is high, the number of records in the discretized database D_f can still be much larger than main memory. (In the worst case, it is the product of the domain sizes of the discretizations of the individual predictor attributes.) Second, a discretization method should take the class label into account (Catlett, 1991a; Fayyad and Irani, 1993) when deciding on the bucket boundaries, but all such discretization methods assume that the input database fits into main memory (Kerber, 1991; Catlett, 1991a; Quinlan, 1993; Fayyad and Irani, 1993; Maass, 1994; Dougherty et al., 1995; Liu and Setiono, 1996; Zighed, 1997). Catlett (1991b) proposed sampling at each node of the classification tree, but considers in his studies only datasets that could fit in main memory. Methods for partitioning the dataset such that each subset fits in main memory are considered by Chan and Stolfo (1993a, b); although this method enables classification of large datasets their studies show that the quality of the resulting classification tree is worse than that of a classifier that was constructed taking the complete database into account at once.

In this paper, we present a framework for scaling up existing classification tree construction algorithms. This general framework, which we call RainForest for rather whimsical reasons,[3] closes the gap between the limitations to main memory datasets of classification tree construction algorithms in the machine learning and statistics literature and the scalability requirements of a data mining environment. The main insight, based on a careful analysis of the algorithms in the literature, is that most (to our knowledge, all) algorithms (including C4.5 (Quinlan, 1993), CART (Breiman et al., 1984), CHAID (Magidson, 1989, 1993a, b), FACT (Loh and Vanichsetakul, 1988), ID3 and extensions (Quinlan, 1979, 1983, 1986; Cheng et al., 1988; Fayyad, 1991), SLIQ and SPRINT (Mehta et al., 1996, 1995; Shafer et al., 1996) and QUEST (Loh and Shih, 1997)) access the data using a common pattern.

We present data access algorithms that scale with the size of the database, adapt gracefully to the amount of main memory available, and are not restricted to a specific tree construction algorithm. (The quality of split selection methods is addressed extensively in statistics and machine learning.) Our framework applied to split selection methods in the literature results in a scalable version of the method *without modifying the result of the method*. Thus, we do not evaluate the quality of the resulting tree, which is not affected by our framework; instead we concentrate on scalability issues. In addition to the generality of our framework, we show that in many common situations our approach offers performance improvements of over a factor of three over SPRINT (Shafer et al., 1996), the fastest scalable classification tree construction algorithm proposed previously.

The rest of the paper is organized as follows. In Section 2, we formally introduce the problems of classification tree construction and describe previous work in the database literature. In Section 3, we introduce our framework and discuss how it encompasses previous work. In Section 4, we present scalable algorithms to construct classification trees, and in Section 5 we present results from a detailed performance evaluation. We conclude in Section 6.

2. Classification trees

In this section, we first introduce some terminology and notation that we will use throughout the paper. Then we state the problem of classification tree construction formally.

2.1. Problem definition

Let $X_1, \ldots, X_m, C$ be random variables where X_i has domain $\mathrm{dom}(\mathrm{X_i})$. Let $P(X', c)$ be a probability distribution on $\mathrm{dom}(\mathrm{X_1}) \times \cdots \times \mathrm{dom}(\mathrm{X_m}) \times \mathrm{dom}(\mathrm{C})$, $X' \subset \mathrm{dom}(\mathrm{X_1}) \times \cdots \times \mathrm{dom}(\mathrm{X_m})$, $c \in \mathrm{dom}(\mathrm{C})$. In terms of the informal introduction in Section 1, the training database D is a random sample from P, the X_i correspond to the predictor attributes and C is the dependent attribute, the class label. A *classifier* is a function $d : \mathrm{dom}(\mathrm{X_1}) \times \cdots \times \mathrm{dom}(\mathrm{X_m}) \mapsto \mathrm{dom}(\mathrm{C})$. Let $t = \langle t \cdot X_1, \ldots, t \cdot X_m, t \cdot C \rangle$ be a record randomly drawn from P. We define the *misclassification rate* $R_d^{\mathtt{Class}}(P)$ of predictor d to be the probability

of predicting the value of the dependent attribute of a new record incorrectly, formally $R_d^{\texttt{Class}}(P) \stackrel{\text{def}}{=} P(d(\langle t \cdot X_1, \ldots, t \cdot X_m \rangle) \neq t \cdot C)$.

A *classification tree* is a special type of classifier. It is a directed, acyclic graph T in the form of a tree. The root of the tree does not have any incoming edges. Every other node has exactly one incoming edge and maybe have two or more outgoing edges. If a node has no outgoing edges it is called a *leaf node*, otherwise it is called an *internal node*. Each internal node n is labeled with one predictor attribute X_n called the *splitting attribute*, each leaf node n is labeled with one class label $c_n \in \text{dom(C)}$. Each edge (n, n') from an internal node n to one of its children n' has a predicate $q_{(n,n')}$ associated with it where $q_{(n,n')}$ involves only the splitting attribute X_n of node n. The set of predicates Q_n on the outgoing edges of an internal node n must be *non-overlapping* and *exhaustive*. A set of predicates Q is *non-overlapping* if the conjunction of any two predicates in Q evaluates to `false`. A set of predicates Q is *exhaustive* if the disjunction of all predicates in P evaluates to `true`. We will call the set of predicates Q_n on the outgoing edges of an internal node n the *splitting predicates of* n; the combined information of splitting attribute and splitting predicates is called the *splitting criteria* of n and is denoted by $\texttt{crit}(n)$.

We associate with each node $n \in T$ a predicate $f_n : \text{dom}(\text{X}_1) \times \cdots \times \text{dom}(\text{X}_\text{m}) \mapsto \{\texttt{true}, \texttt{false}\}$, called its *node predicate*. Informally, f_n is the conjunction of all splitting predicates on the edges of the path from the root node to n. Formally, for the root node n, $f_n \stackrel{\text{def}}{=} \mathit{true}$. For a non-root node n with parent p where $q_{(p,n)}$ is the predicate on the edge from p to n, we define $f_n \stackrel{\text{def}}{=} f_p \wedge q_{(n,p)}$. Since each leaf node $n \in T$ is labeled with one class label $c_n \in \text{dom(C)}$, the leaf node n encodes a classification rule $f_n \rightarrow c_n$. Thus the tree T encodes a function $T : \text{dom}(\text{X}_1) \times \cdots \times \text{dom}(\text{X}_\text{m}) \mapsto \text{dom(C)}$ and is therefore a classifier, called a *classification tree*. (We will denote both the tree as well as the induced predictor by T; the semantics will be clear from the context.) Let us define the notion of the *family of records* of a node in a tree T with respect to a training database D. (We will drop the dependency on D from the notation since it is clear from the context.) For a node $n \in T$ with parent p, F_n is the set of records in D that follows the path from the root to n when being processed by the tree, formally $F_n \stackrel{\text{def}}{=} \{t \in D : f_n(t) = \texttt{true}\}$.

We can now formally state the problem of classification tree construction:

Classification tree construction problem: Given a dataset $D = \{t_1, \ldots, t_n\}$ where the t_i are independent random samples from a probability distribution P, find a classification tree T such that the misclassification rate $R_T^{\texttt{Class}}(P)$ is minimal.

In practice, we use tree construction algorithms that attempt to minimize the misclassification rate through heuristics as discussed in the next section.

A classification tree is usually constructed in two phases. In phase one, the *growth phase*, an overly large classification tree is constructed from the training database. In phase two, the *pruning phase*, the final size of the tree T is determined with the goal being to minimize $R_T^{\texttt{Class}}(P)$.

In general, construction of an optimal tree is very difficult for several different problem definitions, and the problem has been addressed by several authors. Hyafil and Rivest show that construction of the 'smallest' classification tree, namely the tree that requires the least expected number of tests to classify a record, is NP-complete (Hyafil and Rivest, 1976).

Murphy and McCraw prove that for most problem formulations, construction of trees with the least number of nodes is NP-complete (Murphy and McCraw, 1991). Naumov also considered the same problem and showed that it is NP-complete under several optimality measures (Naumov, 1991).

Due to the hardness of obtaining an optimal solution, nearly all classification tree construction algorithms grow the tree top-down in the following greedy way: At the root node, the database is examined and a splitting criterion is selected. Recursively, at a non-root node n, the family of n is examined and from it a splitting criterion is selected. (This is the well-known schema for greedy top-down classification tree induction; for example, a specific instance of this schema for binary splits is shown in Mehta et al. (1996).) This schema is depicted in figure 3. All classification tree construction algorithms that we are aware of proceed according to this schema.

Two different algorithmic issues need to be addressed during the tree growth phase. The first issue is to devise an algorithm such that the resulting tree T models the underlying probability distribution P and (after a possible pruning step) minimizes $R_T^{\texttt{Class}}(P)$; we will call this part of the overall classification tree construction algorithm the *split selection method*. The second issue is to devise an algorithm for data management in the case that the training database does not fit in-memory; we will call this part of the overall classification tree construction algorithm the *data access method*. Note that traditionally it was assumed that the complete training data set would fit in-memory; thus, no special data access method was necessary.

During the pruning phase a third issue arises, namely how to find an estimator $\hat{R}_T^{\texttt{Class}}(P)$ of $R_T^{\texttt{Class}}(P)$ and how to efficiently calculate $\hat{R}_T^{\texttt{Class}}(P)$.

In this paper, we concentrate on the tree growth phase, since it is a very time-consuming part of classification tree construction due to its data-intensive nature (Mehta et al., 1996; Shafer et al., 1996). MDL-based pruning methods are more popular for large datasets (Mehta et al., 1996; Rastogi and Shim, 1998) because they scale well with increasing dataset size. Cross-validation (Breiman et al., 1984) is a popular pruning technique for small training datasets, but it requires construction of several trees from large subsets of the data. Our techniques can also be used to speed up cross-validation for large training datasets in a straightforward manner, but we will not discuss this point further in this paper. How to prune the tree is an orthogonal issue; the techniques that we will present in Section 3 can be combined with any pruning method.

Figure 1 shows an example training database. There are three predictor attributes: Car type, age and number of children. Age and number of children are numerical attributes, whereas car type is a categorical attribute with three categories: sedan, sports utility vehicle and truck. The class label has two categories indicating whether the household subscribes to a certain magazine. An example classification tree is depicted in figure 2.

2.2. Previous work in the database literature

Agrawal et al. introduced an interval classifier that could use database indices to efficiently retrieve portions of the classified dataset using SQL queries (Agrawal et al., 1992). However, the method does not scale to large training sets (Shafer et al., 1996). Fukuda et al. construct

Record Id	Car type	Age	Number of children	Subscription
1	sedan	23	0	yes
2	sports	31	1	no
3	sedan	36	1	no
4	truck	25	2	no
5	sports	30	0	no
6	sedan	36	0	no
7	sedan	25	0	yes
8	truck	36	1	no
9	sedan	30	2	yes
10	sedan	31	1	yes
11	sports	25	0	no
12	sedan	45	1	yes
13	sports	23	2	no
14	truck	45	0	yes

Figure 1. Example input database.

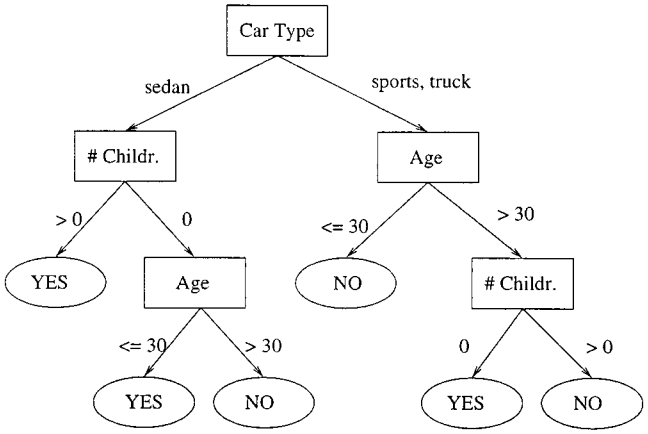

Figure 2. Magazine subscription example classification tree.

classification trees with splitting criteria involving two predictor attributes (Fukuda et al., 1996). Although their algorithm can produce rules with very high classification accuracy, scalability was not one of the design goals. In addition, the classification tree no longer has the intuitive representation of a tree with one-dimensional splits at each node.

The classification tree classifier SLIQ by Mehta et al. is a scalable version of the CART split selection method (Breiman et al., 1984). SLIQ was designed for large training databases but uses an in-memory data structure that grows linearly with the number of records in the training database (Mehta et al., 1996). This limiting data structure was eliminated by Shafer et al. (1996) who introduced SPRINT, a scalable classifier.

SPRINT works for very large datasets and removes all relationships between main memory and size of the training database. SPRINT builds classification trees with binary splits using the `gini` index (Breiman et al., 1984) to decide the splitting criterion; it controls the final quality of the classification tree through an application of the MDL principle (Rissanen, 1989; Mehta et al., 1995). To decide on the splitting attribute at a node n, the algorithm requires access to F_n for each numerical attribute in sorted order. So conceptually, for each node n of the classification tree, a sort of F_n for each numerical attribute is required. SPRINT avoids sorting at each node through the creation of *attribute lists*. The attribute list L_X for attribute X is a vertical partition of the training database D: For each record $t \in D$ the entry of t into L_X consists of the projection of t onto X, the class label and t's record identifier. The attribute lists are created at the beginning of the algorithm and sorted once as a preprocessing step.

During the tree growth phase, whenever an internal node n splits, F_n is distributed among n's children according to `crit`(n). Since every record is vertically partitioned over all attribute lists, each attribute list needs to be distributed across the children separately. The distribution of an attribute list is performed through a hash-join with the attribute list of the splitting attribute. The record identifier, which is duplicated into each attribute list, establishes the connection between the vertical parts of the record. Since during the hash-join each attribute list is read and distributed sequentially, the initial sort order of the attribute list is preserved.

Since the growth phase of SPRINT is essentially a scalable version of the growth phase of the CART algorithm, the techniques in SPRINT could also be used to design a scalable algorithm for regression tree construction. The `gini` index would be replaced with an appropriate criterion for least-squares regression (Breiman et al., 1984).

Morimoto et al. developed algorithms for classification tree construction for categorical predictor attributes with large domains (Morimoto et al., 1998). The goal of their work is to improve the quality of the resulting tree. Rastogi and Shim developed PUBLIC, a MDL-based pruning algorithm for binary trees that is interleaved with the tree growth phase (Rastogi and Shim, 1998). Instead of growing the tree until the leaf nodes contain only records of one class, the authors give conditions when the growth can be stopped while ensuring that the final (pruned) tree is still a subtree of the tree obtained during the growth phase. Since the tree pruning method is orthogonal to tree growth, their techniques can be composed with scalable instantiations of binary split selection methods allowing for a wide applicability of their method, including our work.

2.3. Discussion

Our main contribution is a generic scalable algorithm than can be specialized to obtain scalable versions of most classification and regression tree construction algorithms in the literature. We also improve upon the performance of the only previously known scalable algorithm, SPRINT.

One can think of SPRINT as a *prix fixe* all-you-can-eat meal in a world-class restaurant. SPRINT runs with a minimal amount of main memory and scales to large training databases. But it also comes with some drawbacks. First, it materializes the attribute lists at each node,

possibly tripling the size of the training database (it is possible to create only one attribute list for all categorical attributes together as an optimization). Second, there is a large cost to keep the attribute lists sorted at each node n in the tree: Since the connection between the vertically separated parts of a record can only be made through the record identifier, a costly hash-join needs to be performed. The size of the hash table is proportional to the number of records in F_n and thus can be very large; otherwise several scans are necessary for the join. Overall, SPRINT pays a significant price for its scalability. As we will show in Section 3, some observations about the nature of classification tree construction algorithms enable us to speed up SPRINT significantly in most cases. To return to our restaurant analogy, the techniques in Section 3 allow you to sample some RainForest Crunch (TM) ice-cream in the restaurant, paying for just what you ordered.

The emphasis of the research in the machine learning and statistics community has been on improving the accuracy of classification trees in terms of reducing the misclassification error. Many studies have been performed to determine which algorithm has the highest prediction accuracy (Shavlik et al., 1991; Brodley and Utgoff, 1992; Corruble et al., 1993; Curram and Mingers, 1994; Michie et al., 1994b). These studies show that no algorithm is uniformly most accurate over all the datasets studied. (Mehta et al. also show that the accuracy of the classification tree built by SPRINT is not uniformly superior to other methods (Mehta et al., 1995, 1996).) We have therefore concentrated on developing a unifying framework that can be applied to most classification tree algorithms, and results in a scalable version of the algorithm without modifying the result. That is, *the scalable versions of the algorithms produce exactly the same classification tree as if sufficient main memory were available to run the original algorithm on the complete database in main memory*. To carry our restaurant analogy one (last!) step further, the techniques in Section 4 allow you to pick a different restaurant every day, eat there as little or much as you want, and pay only for what you order.

3. The RainForest framework

We begin in Section 3.1 by introducing the well-known greedy top-down classification tree induction schema. Then we show how this schema can be refined to the generic RainForest Tree Induction Schema and detail how the separation of scalability issues from quality concerns is achieved. After overviewing the resulting design space for the algorithms presented in Section 4, we show a sample instantiation of the framework with the CART algorithm (Breiman et al., 1984) in Section 3.2. We show in Section 3.3 how algorithms for missing values can be easily incorporated into our framework and discuss extensions to regression trees in Section 3.4.

3.1. Classification tree construction

Classification tree algorithms build the tree top-down in the following way: At the root node r, the database is examined and the best splitting criterion `crit`(r) is computed. Recursively, at a non-root node n, F_n is examined and `crit`(n) is computed. (This is the well-known

Input: node n, partition D, split selection method $\mathcal{CL}$
Output: classification tree for D rooted at node n

Top-Down Classification Tree Induction Schema:
BuildTree(Node n, datapartition D, split selection method $\mathcal{CL}$)
(1) Apply $\mathcal{CL}$ to D to find `crit`(n)
(2) Let k be the number of children of n
(3) **if** (k > 0)
(4) Create k children $n_1, \ldots, n_k$ of n
(5) Use best split to partition D into $D_1, \ldots, D_k$
(6) **for** ($i = 1$; $i \leq k$; i++)
(7) BuildTree(n_i, D_i, $\mathcal{CL}$)
(8) **endfor**
(9) **endif**

RainForest Refinement:
(1a) **for** each predictor attribute X
(1b) Call $\mathcal{CL}$.`find_best_partitioning`(AVC-set of X)
(1c) **endfor**
(2a) $\mathcal{CL}$.`decide_splitting_criterion`();
(2b) Let k be the number of children of n

Figure 3. Classification tree induction schema and RainForest refinement.

schema for top-down classification tree induction. For example, a specific instance of this schema for binary trees is shown by Mehta et al. (1996)). This schema is depicted in figure 3.

A thorough examination of the split selection methods in the literature reveals that the greedy schema can be refined to the generic *RainForest Tree Induction Schema* shown in figure 3. Most split selection methods (including C4.5, CART, CHAID, FACT, ID3 and extensions, SLIQ, SPRINT and QUEST) proceed according to this generic schema and we do not know of any algorithm in the literature that does not adhere to it.[4] Consider a node n of the classification tree. The split selection method has to make two classifications while examining the family of n: (i) It has to select the splitting attribute X, and (ii) It has to select the splitting predicates on X. Once decided on the splitting criterion, the algorithm is recursively applied to each of the children of n. In the remainder of the paper, we denote by $\mathcal{CL}$ a representative split selection method.

Note that at a node n, the utility of a predictor attribute X as a possible splitting attribute is examined independent of the other predictor attributes: The *sufficient statistics* is the class label distribution for each distinct attribute value of X. We define the *AVC-set* of a predictor attribute X at node n to be the projection of F_n onto X and the class label where counts of the individual class labels are aggregated. We will denote the AVC-set of predictor attribute

X at node n by $\texttt{AVC}_n(X)$. (The acronym AVC stands for **A**ttribute-**V**alue, **C**lasslabel.) To give a formal definition, assume without loss of gererality that the domain of the class label is the set $\{1, \ldots, J\}$, formally $\text{dom(C)} = \{1, \ldots, \text{J}\}$. Let $a_{n,X,x,i}$ be the number of records t in F_n with attribute value $t \cdot X = x$ and class label $t \cdot C = i$. Formally,

$$a_{n,X,x,i} \stackrel{\text{def}}{=} |\{t \in F_n : t \cdot X = x \wedge t \cdot C = i\}|$$

Let $S \stackrel{\text{def}}{=} \text{dom(X)} \times \mathtt{N}^{\text{J}}$ where $\mathtt{N}$ denotes the set of natural numbers. Then

$$\begin{aligned}\texttt{AVC}_n(X) \stackrel{\text{def}}{=} \{(x, a_1, \ldots, a_J) \in S : \exists t \in F_n \\ : (t \cdot X = x \wedge \forall i \in \{1, \ldots, J\} : a_i = a_{n,X,x,i})\}\end{aligned}$$

We define the *AVC-group* of a node n to be the set of the AVC-sets of all predictor attributes at node n. Note that the size of the AVC-set of a predictor attribute X at node n depends only on the number of distinct attribute values of X and the number of class labels in F_n.

If the training database is stored inside a database system, the AVC-set of a node n for predictor attribute X can be retrieved through a simple SQL-query: `SELECT D.X, D.C, COUNT(*) FROM D WHERE` f_n `GROUP BY D.X, D.C` In order to construct the AVC-sets of all predictor attributes at a node n, a `UNION`-query would be necessary. (In this case, the `SELECT` clause needs to retrieve also the attribute name in order to distinguish individual AVC-sets.) Graefe et al. observe that most database systems evaluate the `UNION`-query through several scans and introduce a new operator that allows gathering of sufficient statistics in one database scan (Graefe et al., 1998).

The main difference between the greedy top-down schema and the subtly refined RainForest Schema is that the latter isolates an important component, the AVC-set. The AVC-set allows the separation of scalability issues of the classification tree construction from the algorithms to decide on the splitting criterion: Consider the main memory requirements at each step of the RainForest Schema shown in figure 3. In lines (1a)–(1c), the AVC-sets of each predictor attribute are needed in main memory, one at a time, to be given as argument to procedure $\mathcal{CL}$.`find_best_partitioning`. Thus, the total main memory required in lines (1a)–(1c) is the maximum size of any single AVC-set. In addition, Algorithm $\mathcal{CL}$ stores for each predictor attribute the result of procedure $\mathcal{CL}$.`find_best_partitioning` as input to the procedure $\mathcal{CL}$.`decide_splitting_criterion`; the size of these statistics is negligible. In line (2a), all the statistics collected in lines (1a)–(1c) are evaluated together in procedure $\mathcal{CL}$.`decide_splitting_criterion`; the main memory requirements for this step are minimal. Lines (3)–(9) distribute records from one partition to several others; one page per open file is needed.

Following the preceding analysis based on insights from the RainForest Schema, we can make the (now rather trivial) observation that as long as we can find an efficient way to

Car type	Subscription	
	Yes	No
sedan	5	2
sports	0	4
truck	1	2

Age	Subscription	
	Yes	No
23	1	1
25	1	2
30	1	1
31	1	1
36	0	3
45	2	0

Num. Children	Subscription	
	Yes	No
0	3	3
1	2	3
2	1	2

Figure 4. AVC-group of the root node for the example input database.

construct the AVC-group of node n, we can scale up any split selection method $\mathcal{CL}$ that adheres to the generic RainForest Schema.

It is worth considering how big the AVC-group at a node n can be; we describe a comparison with the size of an attribute list used in SPRINT in Section 5.3. Consider the size S_X of the AVC-set of predictor attribute X at a node n. Note that S_X is proportional to the number of distinct attribute values of attribute X in F_n, and not to the size of the family F_n of n. Thus, for most real-life datasets, we expect that the whole AVC-group of the root node will fit entirely in main memory, given current memory sizes; if not, it is highly likely that at least the AVC-set of each individual predictor attribute fits in main memory. The assumption that the AVC-group of the root node n fits in-memory does not imply that the input database fits in-memory! The AVC-group of n is *not* a compressed representation of F_n; F_n can not be reconstructed from the AVC-group of n. Rather the AVC-group of n contains aggregated information that is sufficient for classification tree construction. In Section 5, we calculate example numbers for the AVC-group of the root node generated by a synthetic data generator introduced by Agrawal et al. (1993) (which was designed to model real-life data). The maximum memory size for the AVC-group of the generated datasets is about 25 megabytes. With current (December 1998) memory sizes of 128 megabytes for home computers, we believe that in a corporate data mining environment the AVC-group of the root node will almost always fit in main memory; otherwise at least each single AVC-set of the root node will fit in-memory. Figure 4 shows the AVC-sets of the root node from the example input database in figure 1. The AVC-set for predictor attribute car type has three entries, the AVC-set for predictor attribute age has six entries and the AVC-set for predictor attribute number of children has three entries. The AVC-group of the root node consists of all three AVC-sets together.

Depending on the amount of main memory available, three cases can be distinguished:

1. The AVC-group of the root node fits in main memory. We describe algorithms for this case in Sections 4.1, 4.2, and 4.3.
2. Each individual AVC-set of the root node fits in main memory, but the AVC-group of the root node does not fit in main memory. We describe algorithms for this case in Section 4.4.
3. None of the individual AVC-sets of the root fit in main memory. We discuss this case in Section 4.5, and (taking the performance results of Section 5 into account) it is clear

that performance improvements over SPRINT can be expected. However, the algorithm is no longer more general than SPRINT with respect to other classification algorithms.

In understanding the RainForest family of algorithms, it is useful to keep in mind that the following steps are carried out for each tree node n, according to the generic schema in figure 3:

1. *AVC-group construction*: If an AVC-group does not already exist when the node n is considered, we must read F_n in order to construct the AVC-group. This involves a scan of the input database D or a materialized partition of D that is a superset of F_n. Sometimes, we need to construct the AVC-group one AVC-set at a time.
2. *Choose splitting attribute and predicate*: This step uses the split selection method $\mathcal{CL}$ that is being scaled using the RainForest framework; to our knowledge all split selection methods make these choices by examining the AVC-sets of the node one by one.
3. *Partition D across the children nodes*: We must read the entire dataset and write out all records, partitioning them into child "buckets" according to the splitting criterion chosen in the previous step. If there is sufficient memory, we can build the AVC-groups for one or more children at this time, as an optimization.

The algorithms that we present in Section 4 differ primarily in how they utilize additional memory in the third step, and how they deal with insufficient memory to hold an AVC-group in the first step.

3.2. A sample instantiation

In this section we discuss a sample instantiation of our framework with CART, a well-known classification tree construction algorithm (Breiman et al., 1984). (The SPRINT algorithm, the fastest scalable classification algorithm proposed previously, implements the CART split selection method (Shafer et al., 1996).) We concentrate on the CART split selection method and do not discuss cost-complexity pruning, the pruning method of CART, which is orthogonal to its split selection method. (For example, SPRINT uses MDL-pruning instead of cross-validation (Mehta et al., 1995).) CART constructs classification trees with binary splits. Consider an internal node n in the tree with splitting attribute X_n. If X_n is numerical, then the splitting predicates on the outgoing edges of n are of the form $X_n \leq x_n$ and $X_n > x_n$, where $x_n \in \mathrm{dom(X_n)}$. If X_n is categorical, then the splitting predicates on the outgoing edges of n are of the form $X_n \in Y_n$ and $X_n \notin Y_n$, where $Y_n \subset \mathrm{dom(X_n)}$. Since the goal is to separate the class labels through the structure imposed by the classification tree, CART evaluates each potential splitting criterion by calculating the value of an impurity function, the `gini`-index. Assume that a candidate splitting criterion partitions D into D_1 and D_2. The `gini`-index allows to compare quantitatively how "pure" D_1 and D_2 are relative to D.

Consider the example input database in figure 1 which has two class labels. Let p_Y be the relative frequency of class label "Yes" and p_N be the relative frequency of class label "No" in the dataset D. Then the the `gini`-index of the split of a dataset D into D_1 and D_2

is defined as

$$\texttt{gini}(D_1, D_2) \stackrel{\text{def}}{=} \frac{|D_1|}{|D|}\texttt{gini}(D_1) + \frac{|D_2|}{|D|}\texttt{gini}(D_2), \quad \text{where}$$
$$\texttt{gini}(D) \stackrel{\text{def}}{=} 1 - p_Y^2 - p_N^2.$$

For example, the splitting predicate Age ≤ 25 induces a partitioning of D into D_1 and D_2, where D_1 contains the records with record identifiers 1, 4, 7, 11 and 13, D_2 contains the remaining records. Simple calculations show that $\texttt{gini}(D_1, D_2) = 0.49$. While a complete discussion of CART is outside the scope of this paper, the important point is that the arguments to the `gini`-function are the relative frequencies p_Y and p_N of each class label in the partitions D_1 and D_2. These relative frequencies can be easily obtained from the AVC-set of predictor attribute X.

Based on the observations above, we can instantiate our framework with CART as follows. Procedure CART.`find_best_partitioning`, called with an AVC-set of predictor attribute X, computes the (locally) best splitting criterion using predictor attribute X and stores the splitting criterion and the corresponding value of the `gini`-function in-memory. The statistics that are computed by procedure CART.`find_best_partitioning` are examined by CART.`decide_splitting_criterion` which selects the splitting criterion with the global minimum value of the `gini`-function.

3.3. Missing values

It is often the case that real-life training databases have missing values, i.e., in some records one or several predictor attributes have `NULL`-values. Consider a record t with a missing value for predictor attribute X, e.g., $t \cdot X =$ `NULL`, and let n be the root node of the tree. Note that the attribute values $t \cdot X'$ for predictor attributes $X' \neq X$ can be utilized for the computation of the splitting criterion `crit`(n). This utilization is possible since for construction of the AVC-set of predictor attribute X', the projection $\langle t \cdot X', t \cdot C \rangle$ of a record t suffices. Whether t has missing values for predictor attribute X does not influence t's utility for the AVC-sets of predictor attributes $X' \neq X$. But after `crit`(n) has been computed, F_n has to be partitioned among n's children. Assume that X is the splitting attribute at node n. Since for record t, the value of predictor attribute X is missing, it is not clear to which child of n the record t should be sent to. Naively, t can not continue to participate in further construction of the subtree rooted at n. Algorithms for missing values address this problem.

Friedman suggested that all records with missing attribute values should be ignored during tree construction (Friedman, 1977) whereas other authors argue that it is beneficial to make maximum use of all records (Breiman et al., 1984; Loh and Vanichsetakul, 1988). In this section we describe two methods, estimation (Loh and Vanichsetakul, 1988; Loh and Shih, 1997) and surrogate splits (Breiman et al., 1984), and show how they can be incorporated into our framework. The goal of methods is to maximize the use of the training database as well as being able to classify future records with missing values.

In *estimation*, missing values are estimated from the data whenever necessary (Loh and Vanichsetakul, 1988; Loh and Shih, 1997). Assume that at node n we want to estimate the

attribute value of predictor attribute X in record t with class label i, i.e., $t \cdot X = \texttt{NULL}$ and $t \cdot C = i$. (Note that the value of the class label is known for all records in the training dataset.) One simple estimator of $t \cdot X$ is the node class mean, if X is numerical, or the node class mode, if X is categorical. Formally, let $\hat{t} \cdot X$ be the estimated value of $t \cdot X$. Then $\hat{t} \cdot X$ is defined as follows:

$$\hat{t} \cdot X \stackrel{\text{def}}{=} \frac{\sum_{t \in F_n \wedge t \cdot X \neq \texttt{NULL}} t \cdot X}{|\{t \in F_n : t \cdot X \neq \texttt{NULL}\}|}, \qquad \text{if } X \text{ is numerical;}$$

$$\hat{t} \cdot X \stackrel{\text{def}}{=} \texttt{argmax}_{i \in \text{dom(C)}} |\{t \in F_n \wedge t \cdot C = i\}|, \qquad \text{if } X \text{ is categorical.}$$

It can be easily seen that the AVC-set of predictor attribute X contains sufficient statistics to calculate $\hat{t} \cdot X$.

Another method for missing values called *surrogate splits* was introduced by Breiman et al. in CART (Breiman et al., 1984). Let X be the splitting attribute at node n and denote by s_X the splitting criterion at node n. The algorithm selects a splitting criterion $s_{X'}$ for each predictor attribute $X' \neq X$ such that $s_{X'}$ is most similar to s_X in terms of sending records to the same child node. Assume that the best split s_X is on predictor attribute X; s_X partitions F_n into $F_{n_1}, F_{n_2}, \ldots, F_{n_k}$. Consider a split $s_{X'}$ on another predictor attribute X'; $s_{X'}$ partitions F_n into $F_{n'_1}, F_{n'_2}, \ldots, F_{n'_k}$.[5] Then the probability $p(s_X, s_{X'})$ that $s_{X'}$ results in the same prediction as s_X is estimated by the proportion of records in F_n that $s_{X'}$ and s_X send to the same child. Formally,

$$p(s_X, s_{X'}) = \frac{|F_{n_1} \cap F_{n'_1}|}{|F_n|} + \frac{|F_{n_2} \cap F_{n'_2}|}{|F_n|} + \cdots + \frac{|F_{n_k} \cap F_{n'_k}|}{|F_n|}.$$

The best surrogate split for predictor attribute X' with respect to splitting criterion s_X is the split $s_{X'}$ that maximizes $p(s_X, s_{X'})$. (Only surrogate splits that are better than the naive predictor are used. The naive predictor sends a record to the child with the largest family.) Using $p(s_X, s_{X'})$, the predictor attributes $X' \neq X$ can be ranked according to the quality of their surrogate splits as quantified by $p(s_X, s_{X'})$. For a record t with $t \cdot X = \texttt{NULL}$, the prediction of the highest ranked surrogate split $s_{X'}$ such that $t \cdot X' \neq \texttt{NULL}$ is used to determine the child partition of t.

The best surrogate split for a predictor attribute X' is found through an exhaustive search over all possible candidate splits. To calculate the quality of a candidate surrogate split $s_{X'}$, we need the count of records on which the prediction of s_X and $s_{X'}$ agrees. For any candidate split $s_{X'}$, $p(s_X, s_{X'})$ can be calculated from the following data structure: For each attribute value $x \in X'$, we store the count of records that split s_X sends to the ith child of n for $i \in \{1, \ldots, k\}$. These sufficient statistics, which we call the *surrogate-count-set*, are as compact as the AVC-set of predictor attribute X': The size of the surrogate-count-set depends only on the number of different attribute values of predictor attribute X' and the number of partitions induced by s_X. The surrogate-count-set can be created in one scan over the family of n as follows: For each record $t \in F_n$ with $t \cdot X \neq \texttt{NULL}$, we apply s_X. If record t is sent to child number i, the counter for attribute-value $t \cdot X'$ and child number i in the surrogate-count-set of predictor attribute X' is increased. Thus surrogate splits can be easily incorporated into our framework. In case the training database is stored in a database

system, the sufficient statistics for each predictor attribute $X' \neq X$ can be retrieved through a simple SQL-query.

3.4. *Extensions to regression trees*

In this section, we show that our framework can be modified to devise a scalable algorithm to construct regression trees according to the CART regression tree construction algorithm (Breiman et al., 1984). Recall that in a regression problem the dependent variable Y is numerical. A *prediction rule* is defined analogous to a classifier: it is a function $d : \mathrm{dom}(\mathrm{X}_1) \times \cdots \times \mathrm{dom}(\mathrm{X}_\mathrm{m}) \mapsto \mathrm{dom}(\mathrm{Y})$. CART regression trees fit a constant regression model at each leaf node of the tree. We define the *mean squared error* $R_d^{\mathtt{Reg}}(P)$ of prediction rule d to be the expected square error using $d(\langle t \cdot X_1, \ldots, t \cdot X_m \rangle)$ as predictor of $t \cdot Y$, formally $R_d^{\mathtt{Reg}}(P) \stackrel{\mathrm{def}}{=} E(t \cdot Y - d(\langle t \cdot X_1, \ldots, t \cdot X_m \rangle))^2$. A *regression tree* is defined analogous to a classification tree. (The predicted value at each leaf node n is a constant $c_n \in \mathrm{dom}(\mathrm{Y})$.) The *regression tree construction problem* can be formally stated as follows: Given a dataset $D = \{t_1, \ldots, t_n\}$ where the t_i are independent random samples from a probability distribution P, find a regression tree T such that the mean squared error $R_T^{\mathtt{Reg}}(P)$ is minimal (Breiman et al., 1984).

Let $\bar{Y}_n$ be the mean of the dependent variable in node n and R_n be the sum of squares of Y within node n if $\bar{Y}_n$ is used as predictor of Y. Formally,

$$\bar{Y}_n \stackrel{\mathrm{def}}{=} \frac{\sum_{t \in F_n} t \cdot Y}{|F_n|}, \qquad R_n \stackrel{\mathrm{def}}{=} \sum_{t \in F_n} (t \cdot Y - \bar{Y}_n)^2.$$

At node n, let s be a splitting criterion that partitions F_n into $F_{n_1}, \ldots, F_{n_k}$. Then the quality $q(s)$ of splitting criterion s is measured in CART by the weighted decrease in variability. (The function $q(s)$ plays the role of the impurity function discussed in Section 3.2.) Formally:

$$q(s) \stackrel{\mathrm{def}}{=} R_n - \sum_{i=1}^{k} \frac{|F_{n_i}|}{|F_n|} R_{n_i}.$$

Note that R_n can be rewritten as

$$R_n = \sum_{t \in F_n} (t \cdot Y)^2 - |F_n| \cdot \bar{Y}_n^2. \tag{1}$$

Since the splitting criterion $\mathtt{crit}(n)$ involves exactly one predictor attribute, the sufficient statistics to calculate the quality of a candidate split s on predictor attribute X contain the following information: For each attribute value $x \in \mathrm{dom}(\mathrm{X})$ we store the number of tuples t in F_n with $t \cdot X = x$, the sum of the values of the dependent attribute and the sum of squares of the dependent attribute. Formally, let $F_n(X = x) \stackrel{\mathrm{def}}{=} \{t \in F_n : t \cdot X = x\}$. Then for all $x \in \mathrm{dom}(\mathrm{X})$ such that $F_n(X = x) \neq \emptyset$, we store the following information:

$$\vec{x_n} \stackrel{\mathrm{def}}{=} \left(x, |F_n(X = x)|, \sum_{t \in F_n(X=x)} t \cdot Y, \sum_{t \in F_n(X=x)} (t \cdot Y)^2 \right)$$

We define the *AVS-set* of a predictor attribute X at node n as the collection of the values $\vec{x}_n$. (The acronym AVS stands for **A**ttribute-**V**alue, **S**quares.) It is easy to see from Eq. (1) that for any candidate splitting criterion s involving predictor attribute X, its quality $q(s)$ can be calculated from the statistics in the AVS-set. Thus by replacing the AVC-set with the AVS-set, our framework results in a scalable algorithm of the CART regression tree split selection method. In case the training database is stored in a database system, the AVS-set for each predictor attribute X can be retrieved through a simple SQL-query analogous to the query described in Section 3.1.

4. Algorithms

In this section, we present algorithms for two of the three possible relationships between main memory and AVC-set sizes as listed in Section 3. The first three algorithms, RF-Write, RF-Read, and RF-Hybrid, require that the AVC-group of the root node n (and thus the AVC-group of each individual node in the tree) fits into main memory; we assume that this is the most common case, as discussed in Section 3. The remaining algorithm, RF-Vertical, works in the case that each single AVC-set of n fits in-memory, but the complete AVC-group of n does not fit. Since scalability and selection of the splitting criterion are orthogonal in the RainForest Schema, we do not dwell on any issues dealing with the quality of the resulting classification tree. Recall that we denote by $\mathcal{CL}$ a representative split selection methods.

In order to describe the following algorithms precisely, we introduce the notion of the *state* of a node; possible states are `Send`, `Fill`, `FillWrite`, `Write`, `Undecided`, and `Dead`. The state S of a node n determines how a record is processed at n. A list of the states and their preconditions and processing behaviors are shown in figure 5. Whenever a node is created, its state is set to `Undecided` (unless mentioned otherwise), and we will call such a node a *new* node. A node whose state is `Dead` will be called a *dead* node.

State of n	Precondition	Processing Behavior of record t
`Send`	`crit`(n) has been computed; n's children nodes are allocated; n is root node, or parent of n is in state `Send`	t is sent to a child according to `crit`(n)
`Fill`	n is root node, or parent of n is in state `Send`	The AVC-group at n is updated with t
`Write`	n is root node, or parent of n is in state `Send`	t is appended to n's partition
`FillWrite`	n is root node, or parent of n is in state `Send`	The AVC-group at node n is updated by t; t is appended to n's partition
`Undecided`		No processing takes place
`Dead`	`crit`(n) has been computed; either n does not split or all children of n are in state `Dead`	No processing takes place

Figure 5. States and processing behavior.

4.1. *Algorithm RF-Write*

In Algorithm RF-Write we assume that the complete AVC-group of the root node fits into main memory. Let n be the root node of the tree. We make one scan over the database and construct the AVC-group of n. Split selection method $\mathcal{CL}$ is applied and k children nodes of n are created. An additional scan over the database is made, where each record t is written into one of the k partitions. The algorithm then recurses in turn on each partition. In the remainder of this paragraph, we describe Algorithm RF-Write in more detail.

At the beginning, the state of the root node n is set to `Fill` and one scan over the database D is made. Since n is in state `Fill`, its AVC-group is constructed during the scan. Split selection method $\mathcal{CL}$ is called with the AVC-group of n as argument and `crit`(n) is computed. Assume that `crit`(n) splits on predictor attribute X and partitions F_n into k partitions. Algorithm RF-Write allocates k children nodes of n, sets the state of n to `Send`, the state of each child to `Write`, and makes one additional pass over the training database D. Each record t that is read from D is processed by the tree as follows. Since n is in state `Send`, `crit`(n) is applied to t and t is sent to a child node n'. Since node n' is in state `Write`, t is appended to n''s partition. After the scan, the partition of each child node n' consists of $F_{n'}$. The algorithm is then applied on each partition recursively.

For each level of the tree, Algorithm RF-Write reads the entire database twice and writes the entire database once.[6] Note that the construction of the tree does not necessarily have to proceed in a breadth-first or depth-first manner. The only restriction is that we cannot compute the splitting criterion of a node unless the splitting criterion of its parent node has been computed.

4.2. *Algorithm RF-Read*

The basic idea behind Algorithm RF-Read is to always read the original training database instead of writing partitions for the children nodes. Since at some point all AVC-groups of the new nodes will not fit together into main memory, we will read the original database several times, each time constructing AVC-groups for an unexamined subset of the new nodes in the tree.

More precisely, in the first step of Algorithm RF-Read, the state of the root node n is set to `Fill`, one scan over the database D is made, and `crit`(n) is computed. The children nodes $\{n_1, n_2, \ldots, n_k\}$ of n are created. Suppose that at this point there is enough main memory to hold the AVC-groups of all children nodes $\{n_1, n_2, \ldots, n_k\}$ of n in-memory. (We will address the problem of size estimation of the AVC-groups in Section 4.6.) In this case, there is no need to write out partitions for the n_i's as in Algorithm RF-Write. Instead, we can in another scan over D construct the AVC-groups of all children simultaneously: We set the state of n to `Send`, change the state of each newly allocated child n_i from `Undecided` to `Fill`, and build in a second scan over D the AVC-groups of the nodes $n_1, \ldots, n_k$ simultaneously in main memory. After the scan of D, Algorithm $\mathcal{CL}$ is applied to the in-memory AVC-group of each child node n_i to decide `crit`(n_i) for $i \in \{1, \ldots, k\}$. If n_i splits, children nodes of n_i are allocated and n_i's state is set to *Send*; otherwise n_i's state is set to `Dead`. Note that so far we have made only two scans over the training database to construct the first two levels of the tree.

We can proceed in the same way for each level of the tree, as long as there is sufficient main memory available to hold the AVC-groups of all new nodes at the level. Let W be the set of new nodes at level l. Suppose that at a level l there is not sufficient memory to hold all AVC-groups of the new nodes in-memory. In this case, we can divide the set of new nodes W into groups $G_1, \ldots, G_{g_l}$, $\cup G_i = N$, $G_i \cap G_j = \emptyset$ for $i \neq j$, $i, j \in \{1, \ldots, g_l\}$, such that all AVC-groups of the nodes in a given group G_i fit in-memory. Each group is then processed individually: the states of the nodes in G_i are changed from `Undecided` to `Fill` and one scan over the training database is made to construct their AVC-groups. After the scan, their splitting criteria are computed. Once all g_l groups for level l have been processed, we proceed to the next level of the tree. Note that for level l, g_l scans over the training database D were necessary.

With increasing l, usually the number of nodes at the level l of the tree and thus usually the overall main memory requirements of the collective AVC-groups of the nodes at that level grow. Thus, Algorithm RF-Read makes an increasing number of scans over the database per level of the tree. Therefore it is not efficient for splitting algorithms that apply bottom-up pruning (except for the case that the families at the pure leaf nodes are very large—and this is usually not known in advance). But for splitting algorithms that prune the tree top-down (Fayyad, 1991; Rastogi and Shim, 1998), this approach might be a viable solution. In addition, as soon as the family of tuples of a node n fits in-memory, an in-memory classification tree construction algorithm can be run to finish construction of the subtree rooted at n.

We included Algorithm RF-Read for completeness. It marks one end of the design spectrum in the RainForest framework and it is one of the two parents of the Algorithm RF-Hybrid described in the next section. We do not think that it is important in practice.

4.3. Algorithm RF-Hybrid

Combining Algorithm RF-Write and Algorithm RF-Read gives rise to Algorithm RF-Hybrid. We first describe a simple form of RF-Hybrid; in the next paragraph we will refine this version further. RF-Hybrid proceeds exactly like RF-Read until the tree level l is reached where all AVC-groups of the set of new nodes W do not fit any more into main memory collectively. At this point, RF-Hybrid switches to RF-Write: Algorithm RF-Hybrid creates $|W|$ partitions and makes a scan over the database D to distribute D over the $|W|$ partitions. The algorithm then recurses on each node $n \in W$ to complete the subtree rooted at n. This first version of RF-Hybrid uses the available memory more efficiently than RF-Write and, unlike RF-Read, does not require an increasing number of scans over the database for lower levels of the tree.

We can improve upon this simple version of Algorithm RF-Hybrid using the following observation: Assume that we arrive at tree level l where all AVC-groups of the new nodes W together do not fit any more into main memory. Algorithm RF-Hybrid switches from RF-Read to RF-Write, but during this partitioning pass, we do not make use of the available main memory. (Each record is read, processed by the tree and written to a partition—no new information concerning the structure of the tree is gained during this pass.) We exploit this observation as follows. We select a set of nodes $M \subset W$ for which we construct AVC-groups

in main memory while writing the partitions for the nodes in W. After the partitioning pass, split selection method $\mathcal{CL}$ is applied to the in-memory AVC-groups of the nodes in M and their splitting criteria are computed.

The concurrent construction of AVC-groups for the nodes in M has the following advantage. Let $n \in M$ be a node whose AVC-group has been constructed, and consider the recursion of Algorithm RF-Hybrid on n. Since `crit`(n) is already known, we saved the first scan over n's partition. We can immediately proceed to the second scan during which we construct AVC-groups for the children of n. Thus, due to the concurrent construction of the AVC-groups of the nodes in M, we save for each node $n \in M$ one scan over n's partition.

How do we choose $M \subseteq W$? Since we save for each node $n \in M$ one scan over F_n, we would like to maximize the sum of the sizes of the families of the nodes in M. The restricting factor is the size of main memory: For each node $n \in M$ we have to maintain its AVC-group in main memory. We can formulate the problem as follows: Each node $n \in M$ has an associated *benefit* (the size of F_n) and an associated *cost* (the size of its AVC-group which has to be maintained in main memory).

Assume for now that we have estimates of the sizes of the AVC-groups of all nodes in W. (We will address the problem of size estimation of AVC-groups in Section 4.6.) According to the formulation in the preceding paragraph, the choice of M is an instance of the *knapsack problem* (Garey and Johnson, 1979). An instance of the knapsack problem consists of a knapsack capacity and a set of items where each item has an associated cost and benefit. The goal is to find a subset of the items such that the total cost of the subset does not exceed the capacity of the knapsack while maximizing the sum of the benefits of the items in the knapsack. The knapsack problem is known to be NP-complete (Garey and Johnson, 1979). There are both polynomial time approximation schemas and fully polynomial time approximation schemas for knapsack (Ibarra and Kim, 1975; Sahni, 1975). We decided to use a modified greedy approximation which finds a packing that has at least half the benefit of the optimal packing and works well in practice. (We call the greedy algorithm modified, because it considers the item with the largest benefit separately; this special case is necessary to get the stated bound with respect to the optimal solution.) The output of the Greedy Algorithm is the subset M of the new nodes W such that: (i) We can afford to construct the AVC-groups of the nodes in M in-memory, and (ii) the benefit (the number of saved I/O's) is maximized.

4.4. Algorithm RF-Vertical

Algorithm RF-Vertical is designed for the case that the AVC-group of the root node n does not fit in main memory, but each individual AVC-set of n fits in-memory. For the presentation of RF-Vertical, we assume without loss of generality that there are predictor attributes $P_{large} = \{X_1, \ldots, X_v\}$ with very large AVC-sets such that each individual AVC-set fits in main memory, but no two AVC-sets of attributes in P_{large} fit in-memory collectively. We denote the remaining predictor attributes by $P_{small} = \{X_{v+1}, \ldots, X_m\}$. We limited the presentation to this special scenario for the ease of explanation; our discussion can easily be extended to the general case.

State of n	Precondition	Processing Behavior of a record t
`Fill`	n is root node, or the parent of n is in state `Send`	The AVC-group at node n is updated and the projection of t onto P_{large} and the class label is written to a file
`FillWrite`	n is root node, or the parent of n is in state `Send`	The AVC-group at n is updated and t is appended to n's partition; the projection of t onto $Plarge$ and the class label is written to a temporary file F

Figure 6. States with modified processing behavior in RF-Vertical.

Let n be a node in the tree and let t be a record from the database D. In Algorithm RF-Vertical, the processing of t at a node n has slightly changed for some states of n. Assume that n is in state `Fill`. Since we can not afford to construct n's complete AVC-group in main memory, we only construct the attribute lists for predictor attributes P_{small} in-memory. For predictor attributes in P_{large}, we write a temporary file Z_n, into which we insert t's projection onto P_{large} and C. Thus, Z_n has the schema $\langle X_1, X_2, \ldots, X_v, C\rangle$. After the scan over D is completed, procedure $\mathcal{CL}$.`find_best_partitioning` is applied to the in-memory AVC-sets of the attribute in P_{small}. The split selection method $\mathcal{CL}$ can not yet compute the final splitting criterion, since the AVC-sets of the attributes $X \in P_{large}$ have not yet been examined. Therefore, for each predictor attribute $X \in P_{large}$, we make one scan over Z_n, construct the AVC-set for X and call procedure $\mathcal{CL}$.`find_best_partitioning` on the AVC-set. After all v attributes have been examined, we call procedure $\mathcal{CL}$.`decide_splitting_criterion` to compute the final splitting criterion for node n. This slightly modified processing behavior of a node for states `Fill` and `FillWrite` has been summarized in figure 6.

In the description above, we concentrated on one possibility to construct the AVC-set of the predictor attributes P_{large}. In general, there are other possibilities for preparing the construction of the AVC-sets of the predictor attributes P_{large} at a node n. The full set of options is listed below:

1. *Materialize-none*: Each time an AVC-set for a predictor attribute $X \in P_{large}$ needs to be constructed, the dataset D is read. This results in $v \cdot |M|$ additional scans of D, where M is the set of nodes for which AVC-groups are constructed.
2. *Materialize-some*

   (a) *One-file*: This is the algorithm that has been described in the preceding paragraph. At each node n, a file Z_n with schema $\langle X_1, X_2, \ldots, X_v, C\rangle$ is materialized. This materialization takes place during the scan over the dataset D at the root of the current subtree. When a record t arrives at node n, the projection of t onto $\langle X_1, X_2, \ldots, X_v, C\rangle$ is appended to Z_n. To construct the AVC-set of a predictor attribute in P_{large}, Z_n is scanned, resulting in v overall scans of Z_n.

   (b) *Many-files*: At each node n, $l \leq m$ files $Z_n^1, \ldots, Z_n^l$ are materialized. The schema of file Z_n^i is $\langle X_{i_1}, X_{i_2}, \ldots, X_{i_{m_i}}, C\rangle$. If $l = 1$, a single file is materialized and option *many-files* converges to option *one-file*. To construct the AVC-sets of the predictor attributes in P_{large}, file Z_n^i is read m_i times.

3. *Partition*: When a record t arrives at a node n, instead of writing files that contain information about the predictor attributes in P_{large} and the class label, we could as well write the whole record t. In this case, at each node n, one partition $Z_n = F_n$ is materialized which contains the family of records of n. To construct the AVC-sets of the predictor attributes in P_{large} at n, Z is scanned v times. In this case, RF-Vertical writes partitions for each level of the tree and results in a modified version of RF-Write.

4.5. Extensions to AVC-sets larger than main memory

In this section, we discuss the extension of the RainForest framework to AVC-sets larger than main memory. We do so by investigating the relationship between the RainForest family of algorithms and SPRINT (Shafer et al., 1996), resulting in a set of conditions under which RainForest can be applied to input databases with attributes whose AVC-set is larger than main memory.

In the RainForest framework, we make the assumption that the AVC-group of the root node (or at least each single AVC-set of the root node) fits into main memory. This assumption is motivated by the observation that most (to our knowledge, all) classification tree construction algorithms in the literature adhere to the RainForest Tree Induction Schema as outlined in figure 3. Thus, the design goal of the RainForest framework was not to outperform SPRINT, but—based on the assumptions about the relationship between size of main memory and size of an AVC-set—certain optimizations upon SPRINT are possible.

SPRINT achieves its scalability through the creation of attribute lists, which allow *sequential access* to the attribute values of an ordered attribute in *sorted order*—without re-sorting all attribute lists at each node of the tree. Assume that node n splits on splitting attribute X; assume that k children nodes $\{n_1, \ldots, n_k\}$ are created and let $t \in F_n$. In SPRINT, t is vertically partitioned over several attribute lists. The connection between the individual parts is the record identifier, which is duplicated into each attribute list. During distribution of the family of records of n among its children $\{n_1, \ldots, n_k\}$, the individual parts of t need to be re-joined since the entries in the attribute list for an attribute $X' \neq X$ carry no information about the splitting criterion (which involves only attribute X). This join is expensive, but during distribution of an attribute list to the children nodes $\{n_1, \ldots, n_k\}$, the sort order of the attribute list is maintained, thus re-sorting at each node is avoided.

Since Algorithm RF-Hybrid assumes that the AVC-group of the root node fits in-memory, the vertical partitioning of a record is avoided: Each AVC-set is sorted repeatedly in-memory, which is overall much faster than the maintenance of an initial sort of each attribute through the creation of attribute lists. In addition, the partitioning of the family F_n of node n among its children $\{n_1, \ldots, n_k\}$ is a much cheaper operation than in SPRINT: F_n is read once and each record is written to its corresponding child-partition. A similar argument holds for RF-Vertical: RF-Vertical writes vertical partitions for attributes with very large AVC-sets, but uses the partitions only for in-memory construction of the corresponding AVC-set, which is sorted in-memory. Thus F_n is always distributed in whole records among n's children, not in vertical parts.

Note that the assumption that each single AVC-set fits in-memory has implications beyond just performance. Since the AVC-set for attribute X contains all information to compute a

possible splitting criterion on X, the algorithm to compute the splitting criterion can have any access pattern with respect to the AVC-set; SPRINT only provides sequential access in sorted order. Thus for split selection methods that do not exhibit this access pattern (e.g., GID3 and extensions (Fayyad, 1991)), the data management of SPRINT is not applicable. The access pattern generality allows the RainForest framework to scale up a much broader set of algorithms than SPRINT does.

If for a specific classification tree construction algorithm the access pattern that SPRINT provides suffices, it is possible to build a hybrid version between the RainForest algorithms and SPRINT. This hybrid algorithm only creates attribute lists for attributes X whose AVC-set for the root of the tree does not fit in-memory; it maintains the attribute list for X until at a node n the size of X'a AVC-set fits in-memory (which is very likely to happen at a node further down the tree). The idea of maintaining the attribute list sorted through a hash-join carries over from SPRINT. For the remaining attributes, the RainForest optimizations are applied, yielding an efficient algorithm that works for any AVC-set size, but does not exhibit any longer the generality of the RainForest framework.

4.6. AVC-group size estimation

To estimate the size of the AVC-group of new node n, note that we can not assume that n's AVC-group is much smaller than the AVC-group of its parent node p even though F_p might be considerably larger than F_n. Thus, we estimate the size of the AVC-group of a new node n in a very conservative way. We estimate it to be the same size as its parent p—except for the AVC-set of the splitting attribute X. (If parent p of node n splits on X we know the size of X's AVC-set at node n exactly.) This approach usually overestimates the sizes of AVC-sets slightly, but it works very well in practice. There are algorithms for the estimation of the number of distinct values of an attribute (Astrahan et al., 1987; Haas et al., 1995), but we decided not to use these algorithms for the following reason. If we would underestimate the size of an AVC-set, it would have severe consequences, since not sufficient main memory would be available to hold the AVC-set.

5. Experimental results

In the machine learning and statistics literature, the two main performance measures for classification tree algorithms are: (i) The quality of the rules of the resulting tree, and (ii) the classification tree construction time (Lim et al., 1997). The generic schema described in Section 3 allows the instantiation of most (to our knowledge, all) classification tree algorithms from the literature *without modifying the result of the algorithm*. Thus, quality is an orthogonal issue in our framework, and we can concentrate solely on classification tree construction time. In the remainder of this section we study the performance of the techniques that enable classification algorithms to be made scalable.

5.1. Datasets and methodology

The gap between the scalability requirements of real-life data mining applications and the sizes of datasets considered in the literature is especially visible when looking for possible

Predictor Attribute	Distribution	Maximum size
Salary	U(20000,150000)	130001
Commission	If salary > 75k, then commission = 0 else U(10000,75000)	65001
Age	U(20,80)	61
Education level	U(0,4)	5
Car	U(1,20)	20
ZipCode	Uniformly chosen from nine zipcodes	9
House value	U($0.5 \cdot k \cdot 100000$, $1.5 \cdot k \cdot 100000$) where k depends on ZipCode	1350001
Home years	U(1,30)	30
Loan	U(0,500000)	500001
Overall size of the AVC-group of the root		2045129

Figure 7. The sizes of the AVC-sets of the data generator.

benchmark datasets to evaluate scalability results. The largest dataset in the often used Statlog collection of training databases (Michie et al., 1994b) contains only 57000 records, and the largest training dataset considered by Lim, Loh and Shih has 4435 records (Lim et al., 1997). We therefore use the synthetic data generator introduced by Agrawal et al. (1993). The synthetic data has nine predictor attributes as shown in figure 7. Included in the data generator are classification functions that assign labels to the records produced. We show the results of three of the functions (Function 1, Function 6, and Function 7) for our performance study. Functions 1 and 6 generate relatively small classification tree whereas the trees generated by Function 7 are large (Agrawal et al., 1993). (Note that this adheres to the methodology used in the SPRINT and PUBLIC performance studies (Shafer et al., 1996; Rastogi and Shim, 1998).) We ran experiments using other functions in the data generator; the results are qualitatively similar.

Since the feasibility of our framework relies on the size of the initial AVC-group, we examined the size of the AVC-groups of the training data sets generated by the data generator. The overall maximum number of entries in the AVC-group of the root node is about 2.1 million, requiring a maximum memory size of about 25MB. If we partition the predictor attribute *house value* vertically, the main memory requirements to hold the AVC-groups of the root node in main memory are reduced to about 15MB (1.35 million entries). The maximal AVC-set sizes of each predictor attribute are displayed in figure 7. The function $U(x, y)$ denotes the integer uniform distribution with values $v : x \leq v \leq y$. Since we will change the memory available to the RainForest algorithms during our experiments, let us call the number of AVC-set entries that fit in-memory the *buffer size*. So in order to run RF-Write and RF-Hybrid on the datasets generated by the data generator, we need a buffer size of at least 2.1 million entries, whereas RF-Vertical can be run with a buffer size of 1.35 million entries. All our experiments were performed on a Pentium Pro with a 200 Mhz processor running Solaris X86 version 2.6 with 128MB of main memory. All algorithms are written in C++ and were compiled using `gcc` version `pgcc`$-$2.90.29 with the `-O3` compilation option.

We are interested in the behavior of the RainForest algorithms for datasets that are larger than main memory, therefore we uniformly stopped tree construction for leaf nodes whose

family was smaller than 1.5 million records; any clever implementation would switch to a main memory algorithm at a node n whenever F_n fits into main memory.

5.2. *Scalability results*

First, we examined the performance of Algorithms RF-Write, RF-Hybrid and RF-Vertical as the size of the input database increases. For Algorithms RF-Write and RF-Hybrid, we fixed the size of the AVC-group buffer to 2.5 million entries; for Algorithm RF-Vertical we fixed the size of the AVC-group buffer to 1.8 million entries. Figures 8–10 show the overall running time of the algorithms as the number of records in the input database increases from 2 to 10 million. (Function 7 constructs larger trees that branch more often and thus tree growth takes longer than for Functions 1 and 6.) The running time of all algorithms grows nearly linearly with the number of records, as we expected. Algorithm RF-Hybrid outperforms both Algorithms RF-Write and RF-Vertical in terms of running time; the difference is much more pronounced for Function 7. Figures 11–13 show the number of page accesses during tree construction (assuming a pagesize of 128KB).

In the next four experiments, we investigated how internal properties of the AVC-groups of the training database influence performance. (We expected that only the size of the input database and the buffer size matter which is confirmed by the experiments.) We fixed the size of the input database to 6 million records and the sample distribution to Function 1. Figure 14 shows the effect of an increase in the absolute size of the AVC-group in the input database while holding the available buffer sizes constant at 2.5 million entries for RF-Write and RF-Hybrid and at 1.8 million entries for RF-Vertical. We varied the size of

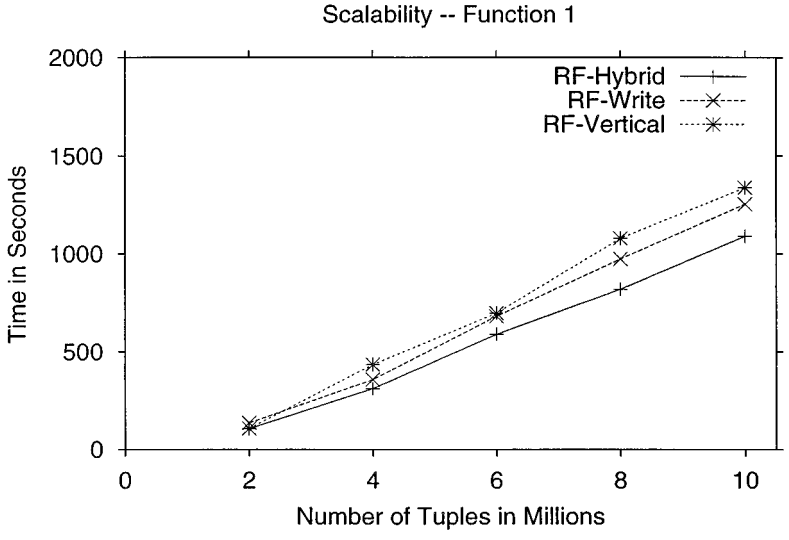

Figure 8. Scalability—F1: Overall time.

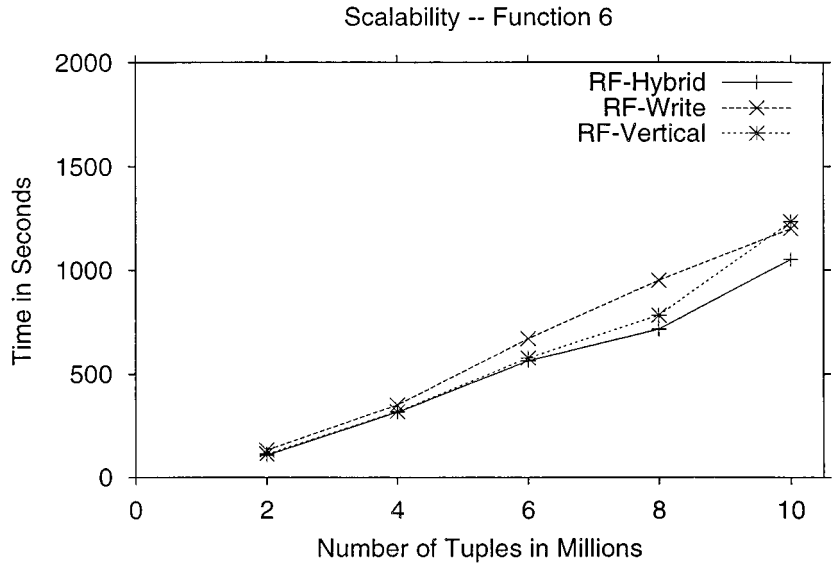

Figure 9. Scalability—F6: Overall time.

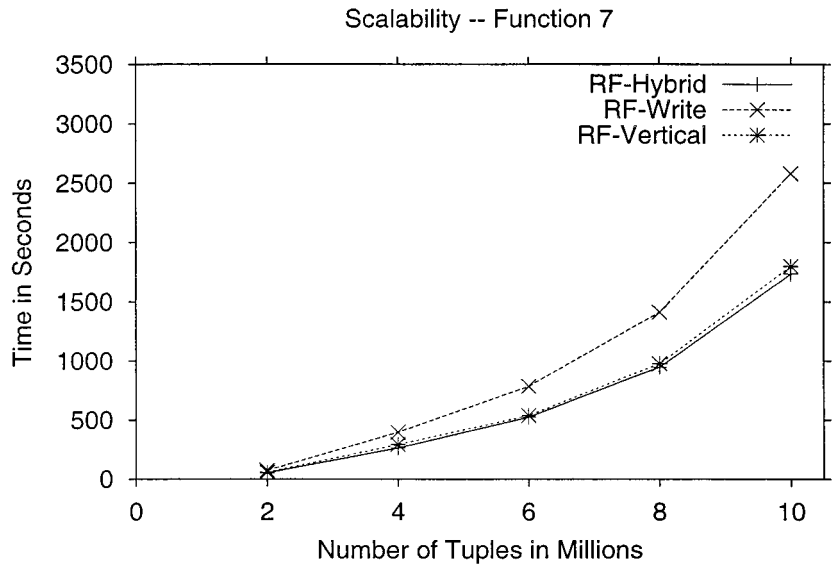

Figure 10. Scalability—F7: Overall time.

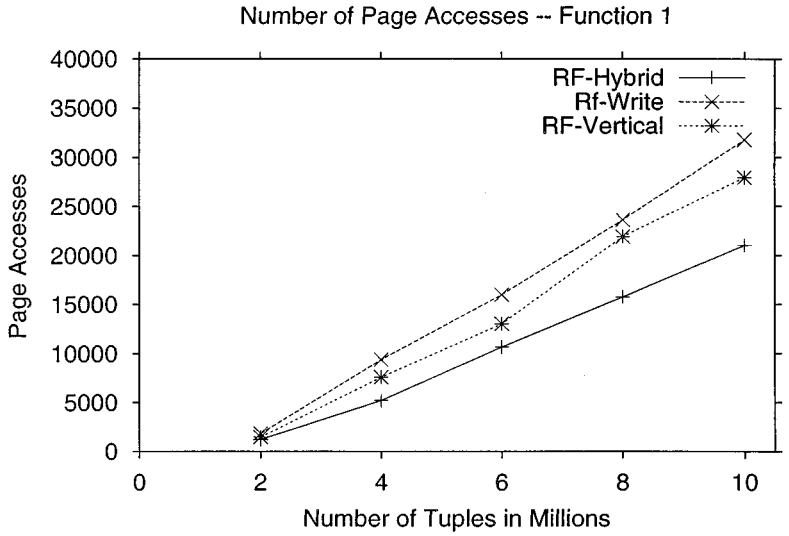

Figure 11. Scalability—F1: Page accesses.

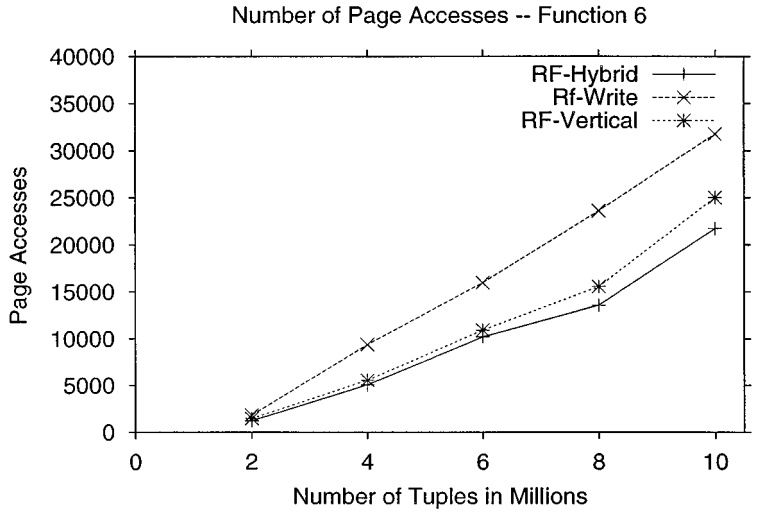

Figure 12. Scalability—F6: Page accesses.

the AVC-group through manipulation of the data generator from 200000 entries (20% of the original size) to 2000000 entries (original size). For small AVC-group sizes (40% and below), the times for RF-Vertical and RF-Hybrid are identical. The larger buffer size only shows its effect for larger AVC-group-sizes: RF-Hybrid writes partitions less frequently than RF-Vertical. The running time of RF-Write is not affected through a change in AVC-group size, since RF-Write writes partitions regardless of the amount of memory available.

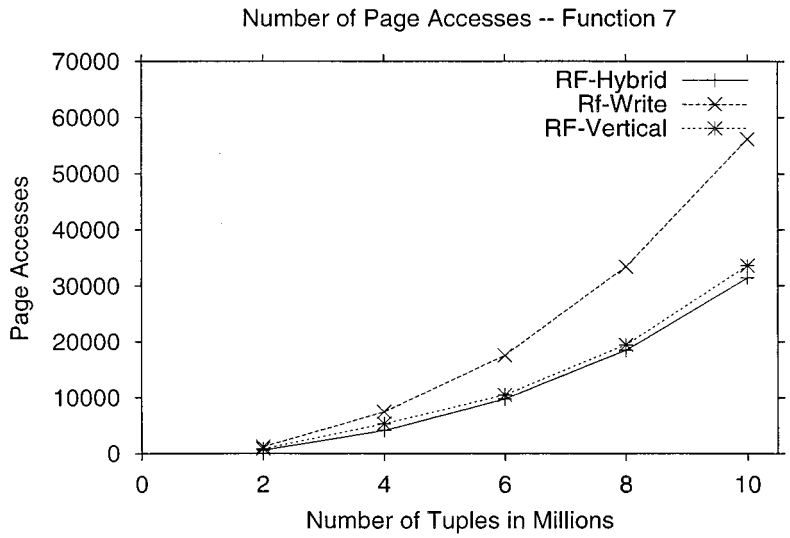

Figure 13. Scalability—F7: Page accesses.

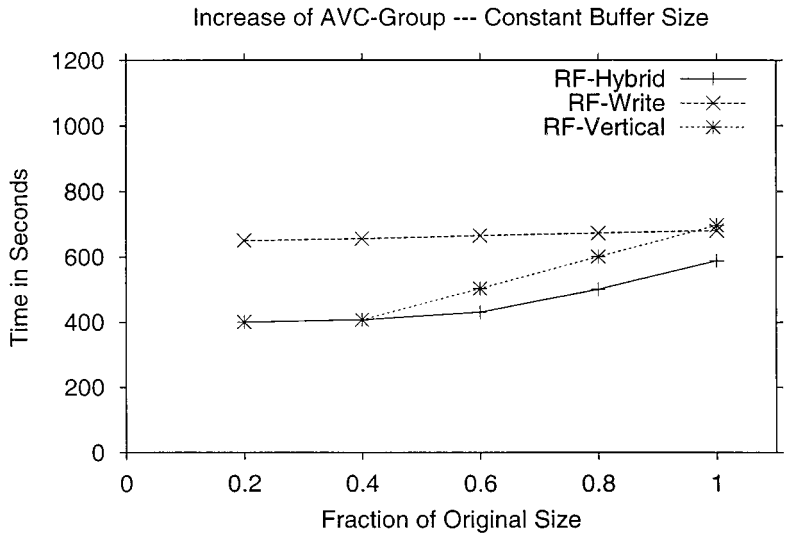

Figure 14. Changing AVC-group size—constant buffer size.

Figure 15 shows the effect of an increase in the absolute size of the AVC-group in the input database while varying the buffer sizes. The buffer size for RF-Write and RF-Hybrid is set such that exactly the AVC-group of the root node fits in-memory; the buffer size of RF-Vertical is set such that exactly the largest AVC-set of the root node fits in-memory.

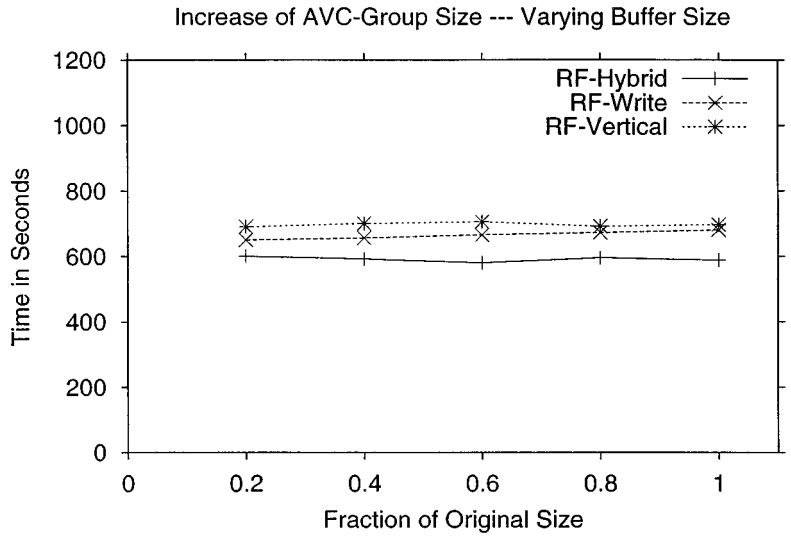

Figure 15. Changing AVC-group size—varying buffer size.

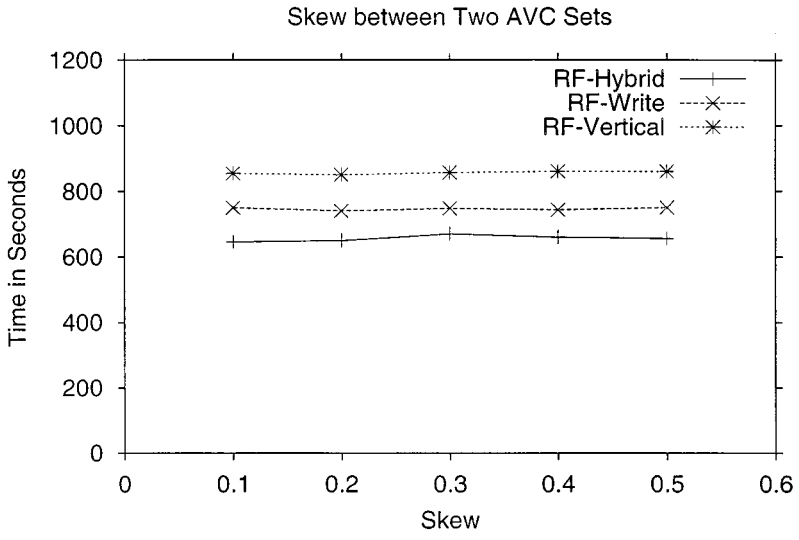

Figure 16. Changing AVC-group skew.

Since both AVC-group size and buffer size are increased simultaneously (keeping their ratio constant), the running times stay constant.

Figure 16 shows how the effect of skew between two attributes within an AVC-group affects performance. The number of records remained constant at 6 million; we set the

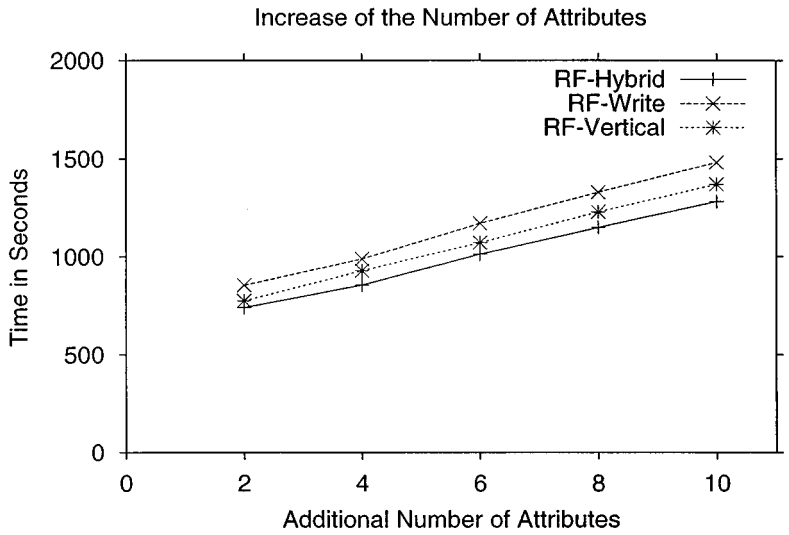

Figure 17. Changing number of attributes.

buffer sizes for RF-Write and RF-Hybrid to 250000, and the buffer size for RF-Vertical to 1800000. We duplicated the `loan` attribute (thus increasing the number of attributes to ten), but skewed the distribution of distinct attributes values between the two loan attributes. We reduced the number of attribute values of the remaining attributes to make the loan attributes the dominant contributors to the overall AVC-group size. While changing the skew, we held the overall number of distinct attribute values for the two loan attributes at a combined size of 1200000 entries. For example, a skew value of 0.1 indicates that the first loan attribute had 10% (120000) distinct attribute values and the second loan attribute had 90% (1080000) distinct values. As we expected, the overall running time is not influenced by the skew, since the overall AVC-group size remained constant.

In our last experiment shown in figure 17, we added extra attributes with random values to the records in the input database, while holding the overall number of entries constant at 4200000 for RF-Hybrid and RF-Write and at 2100000 entries for RF-Vertical. Adding attributes increases tree construction time since the additional attributes need to be processed, but does not change the final classification tree. (The split selection method will never choose such a 'noisy' attribute in its splitting criterion.) As can be seen in figure 17, the RainForest family of algorithms exhibits a roughly linear scaleup with the number of attributes.

5.3. Performance comparison with SPRINT

In this section, we present a performance comparison with SPRINT (Shafer et al., 1996). We tried to make our implementation of SPRINT as efficient as possible, resulting in the

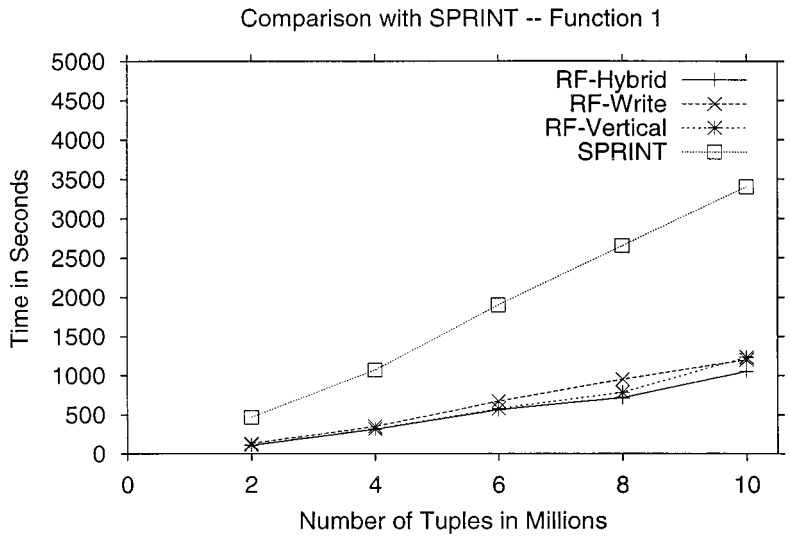

Figure 18. Comparison with SPRINT—F1.

following two implementation improvements over the algorithm described by Shafer et al. (1996). First, we create only one attribute list for all categorical attributes together. Second, when a node n splits into children nodes n_1 and n_2, we create the histograms for the categorical attributes of n_1 and n_2 during the distribution of the categorical attribute list, thus saving an additional scan. We made the in-memory hash-table large enough to perform each hash-join in one pass over an attribute list.

Figures 18–20 show the comparison of SPRINT and the RainForest algorithms for Functions 1, 6, and 7. For algorithms RF-Hybrid and RF-Write, we set the AVC buffer size to 2500000 entries (the AVC-group of the root fits in-memory); for RF-Vertical we set the buffer size such that the largest AVC-set of a single attribute of the root node fits in-memory. The figures show that the RainForest algorithms outperform SPRINT by about a factor of three.

Where does this speed-up come from? First, we compared the cost of the repeated in-memory sorting of AVC-groups in the RainForest algorithms with the cost of creation of attribute lists in SPRINT through which repeated sorts can be avoided. The numbers in figure 21 show that repeated in-memory sorting of the AVC-groups is about ten times faster than the initial attribute list creation time. Second, we compared the cost to arrive at a splitting criterion for a node n plus distribution of F_n among n's children. In SPRINT, the splitting criterion is computed through a scan over all attribute lists; the distribution of F_n is performed through a hash-join of all attribute lists with the attribute list of the splitting attribute. In the RainForest family of algorithms, F_n is read twice and written once; RF-Vertical needs to write vertical partitions if necessary. (We forced the algorithms to write partitions to compare the performance to SPRINT.) We set the buffer size of RF-Write such that the AVC-group of the root fits in-memory and the buffer size of RF-Vertical

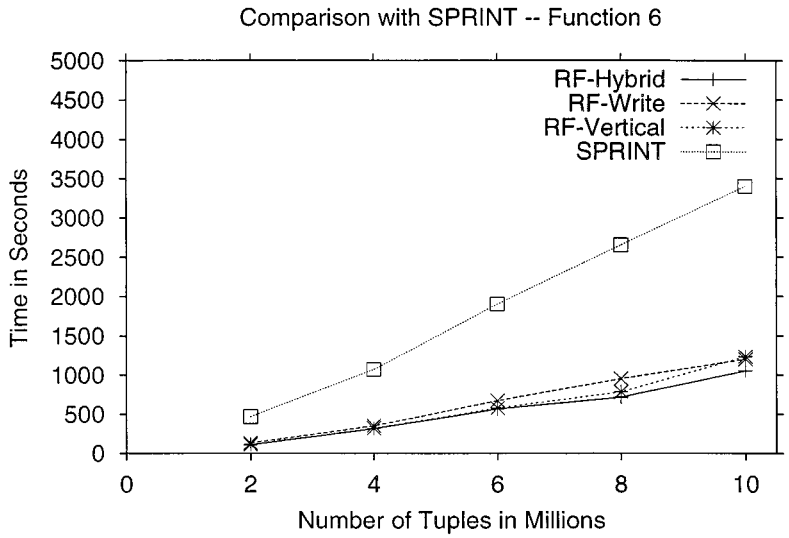

Figure 19. Comparison with SPRINT—F6.

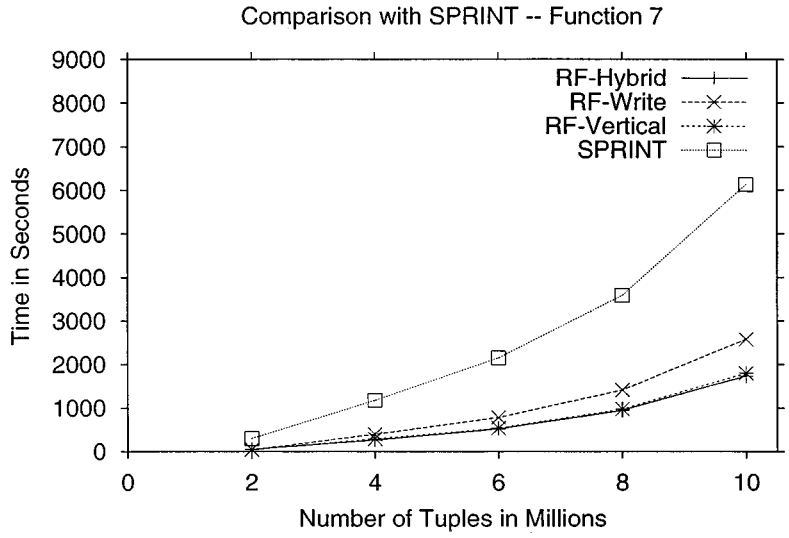

Figure 20. Comparison with SPRINT—F7.

such that the largest AVC-set fits in-memory. Figure 22 shows that the cost of determining the splitting criterion plus partitioning in the original database is about a factor of three faster than scanning and hash-joining the attribute lists. This cost is the overall dominant cost during tree construction and thus explains why the RainForest family of algorithms outperforms SPRINT by a factor of three.

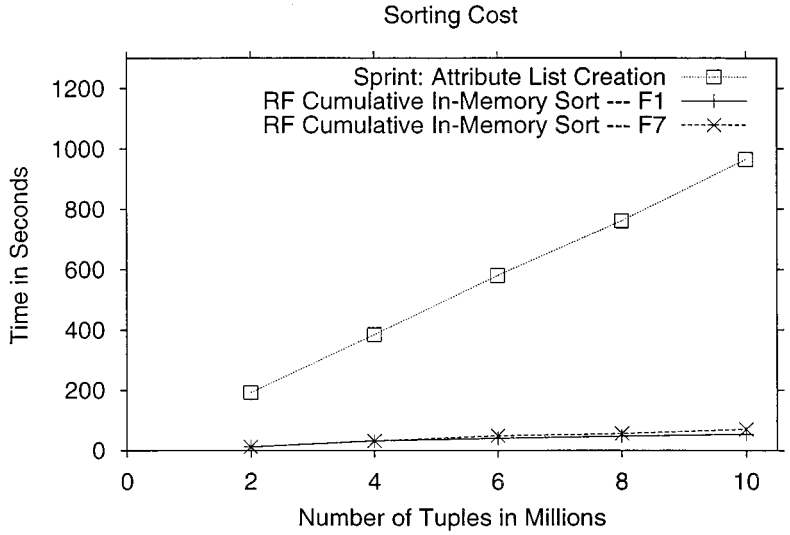

Figure 21. Sorting cost comparison.

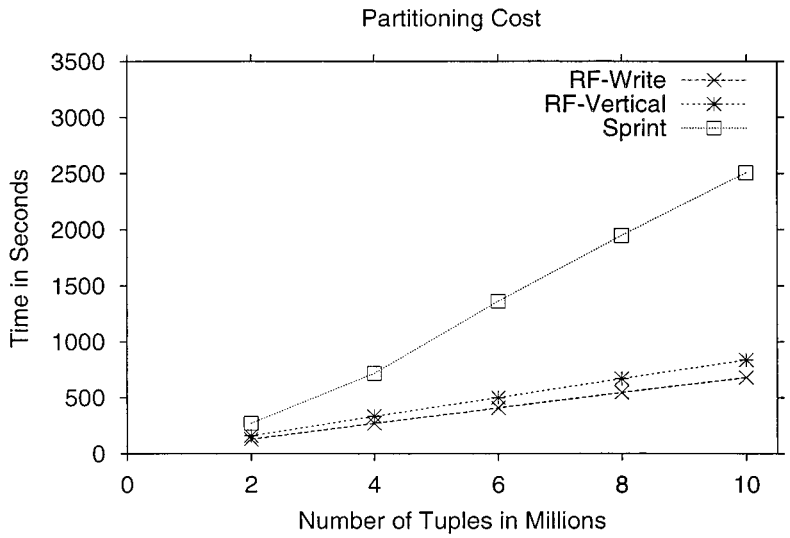

Figure 22. Partitioning cost comparison.

6. Conclusions

In this paper, we have developed a comprehensive approach to scaling classification tree algorithms that is applicable to all classification tree algorithms that we are aware of. The

key insight is the observation that classification trees in the literature base their splitting criteria at a tree node on the AVC-group for that node, which is relatively compact.

The best splitting criteria developed in statistics and machine learning can now be exploited for classification in a scalable manner. In addition, depending upon the available memory, our algorithms offer significant performance improvements over the SPRINT classification algorithm, which is the fastest scalable classifier in the literature. If there is enough memory to hold individual AVC-sets, as is very likely, we obtain very good speed-up over SPRINT; if there is enough memory to hold all AVC-sets for a node, the speed-up is even better.

Acknowledgments

We thank Wei-Yin Loh for insightful discussions. We thank Ramakrishnan Srikant for providing us with the data generator. We thank Sreerama K. Murthy for providing his classification tree bibliography.

Notes

1. A *categorical* attribute takes values from a set of categories. Some authors distinguish between categorical attributes that take values in an unordered set (*nominal* attributes) and categorical attributes having ordered scales (*ordinal* attributes).
2. Classification trees are often also called *decision trees*. In this paper, we refer to tree-structured models for both regression and classification problems as *decision trees*, since each node in the tree encodes a decision.
3. There are lots of trees to choose from, and they all grow fast in RainForest! We also happen to like rainforests.
4. Note that we consider classification trees whose splitting criteria involve a single predictor attribute. Our schema does not encompass split selection methods that result in splitting criteria involving several predictor attributes, such as the work by Fukuda et al. (1996).
5. Breiman et al. only consider binary trees, although the technique generalizes to k-ary trees.
6. This simple analysis assumes that the tree is balanced. More precisely, at a level l, only those records that belong to families of nodes at level l are read twice and written once. Since there might be dead nodes in the tree, the set of records processed at level l does not necessarily constitute the whole training database.

References

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

Agresti, A. 1990. Categorical Data Analysis. John Wiley and Sons.

Astrahan, M.M., Schkolnick, M., and Whang, K.-Y. 1987. Approximating the number of unique values of an attribute without sorting. Information Systems, 12(1):11–15.

Brachman, R.J., Khabaza, T., Kloesgen, W., Shapiro, G.P., and Simoudis, E. 1996. Mining business databases. Communications of the ACM, 39(11):42–48.

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

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

Brodley, C.E. and Utgoff, P.E. 1992. Multivariate versus univariate decision trees. Technical Report 8, Department of Computer Science, University of Massachussetts, Amherst, MA.

Catlett, J. 1991a. On changing continuos attributes into ordered discrete attributes. Proceedings of the European Working Session on Learning: Machine Learning, 482:164–178.

Catlett, J. 1991b. Megainduction: Machine learning on very large databases. PhD Thesis, University of Sydney.

Chan, P.K. and Stolfo, S.J. 1993a. Experiments on multistrategy learning by meta-learning. In Proc. Second Intl. Conference on Info. and Knowledge Mgmt., pp. 314–323.

Chan, P.K. and Stolfo, S.J. 1993b. Meta-learning for multistrategy and parallel learning. In Proc. Second Intl. Workshop on Multistrategy Learning, pp. 150–165.

Cheeseman, P. and Stutz, J. 1996. Bayesian classification (autoclass): Theory and results. In Advances in Knowledge Discovery and Data Mining, U.M. Fayyad, G.P. Shapiro, P. Smyth, and R. Uthurusamy (Eds.). AAAI/MIT Press, ch. 6, pp. 153–180.

Cheeseman, P., Kelly, J., Self, M., Stutz, J., Taylor, W., and Freeman, D. 1988. Autoclass: A bayesian classification system. In Proceedings of the Fifth International Conference on Machine Learning. Morgan Kaufmann.

Cheng, J., Fayyad, U.M., Irani, K.B., and Qian, Z. 1988. Improved decision trees: A generalized version of ID3. In Proceedings of the Fifth International Conference on Machine Learning. Morgan Kaufman.

Chirstensen, R. 1997. Log-Linear Models and Logistic Regression, 2nd ed. Springer.

Corruble, V., Brown, D.E., and Pittard, C.L. 1993. A comparison of decision classifiers with backpropagation neural networks for multimodal classification problems. Pattern Recognition, 26:953–961.

Curram, S.P. and Mingers, J. 1994. Neural networks, decision tree induction and discriminant analysis: An empirical comparison. Journal of the Operational Research Society, 45:440–450.

Dougherty, J., Kahove, R., and Sahami, M. 1995. Supervised and unsupervised discretization of continous features. In Machine Learning: Proceedings of the 12th International Conference, A. Prieditis and S. Russell (Eds.). Morgan Kaufmann.

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

Fayyad, U., Haussler, D., and Stolorz, P. 1996. Mining scientific data. Communications of the ACM, 39(11).

Fayyad, U.M. and Irani, K. 1993. Multi-interval discretization of continous-valued attributes for classification learning. In Proceedings of the 13th International Joint Conference on Artificial Intelligence. Morgan Kaufmann, pp. 1022–1027.

Fayyad, U.M., Shapiro, G.P., Smyth, P., and Uthurusamy, R. (Eds.). 1996. Advances in Knowledge Discovery and Data Mining. AAAI/MIT Press.

Friedman, J.H. 1977. A recursive partitioning decision rule for nonparametric classifiers. IEEE Transactions on Computers, 26:404–408.

Fukuda, T., Morimoto, Y., and Morishita, S. 1996. Constructing efficient decision trees by using optimized numeric association rules. In Proceedings of the 22nd VLDB Conference. Mumbai, India.

Garey, M.R. and Johnson, D.S. 1979. Computer and Intractability. Freeman and Company.

Gillo, M.W. 1972. MAID: A honeywell 600 program for an automatised survey analysis. Behavioral Science, 17:251–252.

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

Graefe, G., Fayyad, U., and Chaudhuri, S. 1998. On the efficient gathering of sufficient statistics for classification from large SQL databases. In Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining. AAAI Press, pp. 204–208.

Haas, P.J., Naughton, J.F., Seshadri, S., and Stokes, L. 1995. Sampling-based estimation of the number of distinct values of an attribute. In Proceedings of the Eighth International Conference on Very Large Databases (VLDB). Zurich, Switzerland, pp. 311–322.

Hand, D.J. 1997. Construction and Assessment of Classification Rules. Chichester, England: John Wiley & Sons.

Hyafil, L. and Rivest, R.L. 1976. Constructing optimal binary decision trees is NP-complete. Information Processing Letters, 5(1):15–17.

Ibarra, O.H. and Kim, C.E. 1975. Fast approximation algorithms for the knapsack and sum of subsets problem. Journal of the ACM, 22:463–468.

Inman, W.H. 1996. The data warehouse and data mining. Communications of the ACM, 39(11).

James, M. 1985. Classification Algorithms. Wiley.

Kerber, R. 1991. Chimerge discretization of numeric attributes. In Proceedings of the 10th International Conference on Artificial Intelligence, pp. 123–128.

Kohavi, R. 1995. The power of decision tables. In Proceedings of the 8th European Conference on Machine Learning. N. Lavrac and S. Wrobel (Eds.). Lecture Notes in Computer Science, vol. 912, Springer.

Kohonen, T. 1995. Self-Organizing Maps. Heidelberg: Springer-Verlag.

Lim, T.-S., Loh, W.-Y., and Shih, Y.-S. 1997. An empirical comparison of decision trees and other classification methods. Technical Report 979, Department of Statistics, University of Wisconsin, Madison.

Liu, H. and Setiono, R. 1996. Chi2: Feature selection and discretization of numerical attributes. In Proceedings of the IEEE Tools on AI.

Loh, W.-Y. and Shih, Y.-S. 1997. Split selection methods for classification trees. Statistica Sinica, 7(4):815–840.

Loh, W.-Y. and Vanichsetakul, N. 1988. Tree-structured classification via generalized disriminant analysis (with discussion). Journal of the American Statistical Association, 83:715–728.

Maass, W. 1994. Efficient agnostic pac-learning with simple hypothesis. In Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory, pp. 67–75.

Magidson, J. 1989. CHAID, LOGIT and log-linear modeling. Markting Information Systems, Report 11-130.

Magidson, J. 1993a. The CHAID approach to segmentation modeling. In Handbook of Marketing Research, R. Bagozzi (Ed.). Blackwell.

Magidson, J. 1993b. The use of the new ordinal algorithm in CHAID to target profitable segments. Journal of Database Marketing, 1(1).

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

Mehta, M., Rissanen, J., and Agrawal, R. 1995. MDL-based decision tree pruning. In Proc. of the 1st Int'l Conference on Knowledge Discovery in Databases and Data Mining, Montreal, Canada.

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

Michie, D., Spiegelhalter, D.J., and Taylor, C.C. (Eds.). 1994b. Machine Learning, Neural and Statistical Classification. London: Ellis Horwood.

Morgan, J.N. and Messenger, R.C. 1973. Thaid: A sequantial search program for the analysis of nominal scale dependent variables. Technical Report, Institute for Social Research, University of Michigan, Ann Arbor, Michigan.

Morimoto, Y., Fukuda, T., Matsuzawa, H., Tokuyama, T., and Yoda, K. 1998. Algorithms for mining association rules for binary segmentations of huge categorical databases. In Proceedings of the 24th International Conference on Very Large Databases (VLDB). Morgan Kaufmann.

Murphy, O.J. and McCraw, R.L. 1991. Designing storage efficient decision trees. IEEE Trans. on Comp., 40(3):315–319.

Murthy, S.K. 1995. On growing better decision trees from data. PhD Thesis, Department of Computer Science, Johns Hopkins University, Baltimore, Maryland.

Naumov, G.E. 1991. NP-completeness of problems of construction of optimal decision trees. Soviet Physics, Doklady, 36(4):270–271.

Quinlan, J.R. 1979. Discovering rules by induction from large collections of examples. In Expert Systems in the Micro Electronic Age, D. Michie (Ed.). Edinburgh University Press: Edinburgh, UK.

Quinlan, J.R. 1983. Learning efficient classification procedures. In Machine Learning: An Artificial Intelligence Approach, T.M. Mitchell, R.S. Michalski, and J.G. Carbonell (Eds.). Palo Alto, CA: Tioga Press.

Quinlan, J.R. 1986. Induction of decision trees. Machine Learning, 1:81–106.

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 Databases. New York City, New York, pp. 404–415.

Ripley, B.D. 1996. Pattern Recognition and Neural Networks. Cambridge: Cambridge University Press.

Rissanen, J. 1989. Stochastic Complexity in Statistical Inquiry. World Scientific Publ. Co.

Sahni, S. 1975. Approximate algorithms for the 0/1 knapsack problem. Journal of the ACM, 22:115–124.

Sarle, W.S. 1994. Neural networks and statistical models. In Procedings of the Nineteenth Annual SAS Users Groups International Conference. SAS Institute, Inc., Cary, NC, pp. 1538–1550.

Shafer, J., Agrawal, R., and Mehta, M. 1996. SPRINT: A scalable parallel classifier for data mining. In Proc. of the 22nd Int'l Conference on Very Large Databases. Bombay, India.

Shavlik, J.W., Mooney, R.J., and Towell, G.G. 1991. Symbolic and neural learning algorithms: An empirical comparison. Machine Learning, 6:111–144.

Sonquist, J.A., Baker, E.L., and Morgan, J.N. 1971. Searching for structure. Technical Report, Institute for Social Research, University of Michigan, Ann Arbor, Michigan.

Weiss, S.M. and Kulikowski, C.A. 1991. Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems. Morgan Kaufman.

Zighed, D.A., Rakotomalala, R., and Feschet, F. 1997. Optimal multiple intervals discretization of continous attributes for supervised learning. In Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, pp. 295–298.