BOAT—Optimistic Decision Tree Construction

Johannes Gehrke[*] *Venkatesh Ganti* *Raghu Ramakrishnan*[†] *Wei-Yin Loh*[‡]
Department of Computer Sciences and Department of Statistics
University of Wisconsin-Madison

Abstract

Classification is an important data mining problem. Given a training database of records, each tagged with a class label, the goal of classification is to build a concise model that can be used to predict the class label of future, unlabeled records. A very popular class of classifiers are decision trees. All current algorithms to construct decision trees, including all main-memory algorithms, make one scan over the training database per level of the tree.

We introduce a new algorithm (BOAT) for decision tree construction that improves upon earlier algorithms in both performance and functionality. BOAT constructs several levels of the tree in only two scans over the training database, resulting in an average performance gain of 300% over previous work. The key to this performance improvement is a novel *optimistic* approach to tree construction in which we construct an initial tree using a small subset of the data and refine it to arrive at the final tree. We guarantee that any difference with respect to the "real" tree (i.e., the tree that would be constructed by examining all the data in a traditional way) is detected and corrected. The correction step occasionally requires us to make additional scans over subsets of the data; typically, this situation rarely arises, and can be addressed with little added cost.

Beyond offering faster tree construction, BOAT is the first scalable algorithm with the ability to incrementally update the tree with respect to both insertions and deletions over the dataset. This property is valuable in dynamic environments such as data warehouses, in which the training dataset changes over time. The BOAT update operation is much cheaper than completely re-building the tree, and the resulting tree is guaranteed to be identical to the tree that would be produced by a complete re-build.

1 Introduction

Classification is an important data mining problem. The input is a dataset of *training records* (also called *training database*), 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 *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.) The goal of classification is to build a concise model of the distribution of the class label in terms of the predictor attributes. The resulting model is used to assign class labels to future records where the values of the predictor attributes are known but the value of the class label is unknown. Classification has a wide range of applications, including scientific experiments, medical diagnosis, fraud detection, credit approval, and target marketing [FPSSU96].

Many classification models have been proposed in the literature [MST94, Han97]. Classification trees [BFOS84], also called *decision 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 [BFOS84, MAR96]. Second, decision trees do not require any parameter setting from the user and thus are especially suited for exploratory knowledge discovery. Third, decision trees can be constructed relatively fast compared to other methods [SAM96, GRG98]. Last, the accuracy of decision trees is comparable or superior to other classification models [Mur95, LLS97]. In this paper, we restrict our attention to decision trees.

We introduce a new algorithm called `BOAT`[2] that addresses both performance and functionality issues of decision tree construction. `BOAT` outperforms existing algorithms by a factor of three while constructing exactly the same decision tree; the increase in speed does not come at the expense of quality. The key to this performance improvement is a novel optimistic approach to tree construction, in which statistical techniques are exploited to construct the tree based on a small subset of the data, while guaranteeing that any difference with respect to the "real" tree (i.e., the tree that would be constructed by examining all the data) is detected and corrected by a subsequent scan of all the data. If correction is not possible, the affected part of the tree needs to be processed again; typically, this case rarely arises, or only affects a small portion of the tree and can be addressed with little added cost.

In addition, `BOAT` enhances functionality over previous methods in two major ways. First, `BOAT` is the first scalable algorithm that can maintain a decision tree incrementally

[*] Supported by an IBM Graduate Fellowship
[†] Supported by Grant 2053 from the IBM Corporation.
[‡] Supported by ARO grant DAAG55-98-1-0333.

[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] **B**ootstrapped **O**ptimistic **A**lgorithm for **T**ree Construction

when the training dataset changes dynamically. For example, in a credit card company new transactions arrive continuously; it is crucial that a decision tree based fraud detection system can reflect the most recent fraudulent transactions. One possibility is to rebuild the tree from time to time, e.g., every night, which is an expensive operation. Instead of rebuilding, BOAT allows us to "update" the current tree to incorporate new training data while maintaining the best tree. That is, the tree resulting from the update operation is exactly the same tree as if a traditional algorithm were run on the modified training database. The update operation in BOAT is much cheaper than a complete rebuild of the tree.

Second, BOAT greatly reduces the number of database scans, and is thus well-suited for training databases defined through complex queries over a data warehouse, as long as random samples from parts of the training database can be obtained. (It is possible to obtain a random sample for a broad class of queries [Olk93].) For example, in a data-warehousing environment, BOAT enables mining of decision trees from any star-join query without materializing the training set. All previous algorithms need the training database to be materialized to run efficiently. In addition, in most cases, BOAT does not write any temporary data structures on secondary storage, and thus has low run-time resource requirements.

The remainder of the paper is structured as follows. In Section 2, we formally introduce the problem of decision tree construction. We present our new algorithm in Section 3 and show it can be extended to work in a dynamic environment in Section 4. We present results from a performance evaluation in Section 5. We survey existing work on decision tree classifiers in Section 6 and conclude in Section 7.

2 Preliminaries

2.1 Problem Definition

In this section, we first introduce some terminology and notation that we will use throughout the paper. We then state the problem formally and give some background knowledge about decision tree construction. Let $X_1, \dots, X_m, C$ be random variables where X_i has domain $\mathtt{dom}(X_i)$; we assume without loss of generality that $\mathtt{dom}(C) = \{1, 2, \dots, k\}$. A *classifier* is a function $d : \mathtt{dom}(X_1) \times \cdots \times \mathtt{dom}(X_m) \mapsto \mathtt{dom}(C)$. Let $P(X', C')$ be a probability distribution on $\mathtt{dom}(X_1) \times \cdots \times \mathtt{dom}(X_m) \times \mathtt{dom}(C)$ and denote by $t = \langle t.X_1, \dots, t.X_m, t.C\rangle$ a record randomly drawn from P, i.e., t has probability $P(X', C')$ that $\langle t.X_1, \dots, t.X_m\rangle \in X'$ and $t.C \in C'$. We define the *misclassification rate* R_d of classifier d to be $P(d(\langle t.X_1, \dots, t.X_m\rangle) \neq t.C)$. In terms of the informal introduction, the training database D is a random sample from P, the X_i correspond to the predictor attributes and C is the class label attribute.

A *decision 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 may have outgoing edges. In this research, we concentrate on binary decision trees, since they are the most popular class of decision trees; our techniques can be generalized to non-binary decision trees, although we will not address that case in this paper. Thus we assume in the remainder of this paper that each node has either zero or two outgoing edges. If a node has no outgoing edges it is called a *leaf node*, otherwise it is called an *internal node*. Each leaf node is labeled with one class label; each internal node n is labeled with one predictor attribute X_n called the *splitting attribute*. Each internal node n has a predicate q_n, called the *splitting predicate* associated with it. If X_n is a numerical attribute, q_n is of the form $X_n \leq x_n$, where $x_n \in \mathtt{dom}(X_n)$; x_n is called the *split point* at node n. If X_n is a categorical attribute, q_n is of the form $X_n \in Y_n$ where $Y_n \subset \mathtt{dom}(X_n)$; Y_n is called the *splitting subset* at node n. The combined information of splitting attribute and splitting predicates at node n is called the *splitting criterion* of n. If we talk about the splitting criterion of a node n in the final tree that is output by the algorithm, we sometimes refer to it as the *final splitting criterion*. We will use the terms *final splitting attribute*, *final splitting subset*, and *final split point*, analogously.

We associate with each node $n \in T$ a predicate $f_n : \mathtt{dom}(X_1) \times \cdots \times \mathtt{dom}(X_m) \mapsto \{\mathtt{true}, \mathtt{false}\}$, called its *node predicate* as follows: for the root node n, $f_n \stackrel{\text{def}}{=} true$. Let n be a non-root node with parent p whose splitting predicate is q_p. If n is the left child of p, define $f_n \stackrel{\text{def}}{=} f_p \wedge q_p$; if n is the right child of p, define $f_n \stackrel{\text{def}}{=} f_p \wedge \neg q_p$. Informally, f_n is the conjunction of all splitting predicates on the internal nodes on the path from the root node to n. Since each leaf node $n \in T$ is labeled with a class label, it encodes a classification rule $f_n \rightarrow c$, where c is the label of n. Thus the tree T encodes a function $T : \mathtt{dom}(X_1) \times \cdots \times \mathtt{dom}(X_m) \mapsto \mathtt{dom}(C)$ and is therefore a classifier, called a *decision tree classifier*. (We will denote both the tree as well as the induced classifier by T; the semantics will be clear from the context.) Let us define the notion of the *family of tuples* of a node in a decision tree T with respect to a 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 $F_n \stackrel{\text{def}}{=} \{t \in D : f_n(t) = \mathtt{true}\}$. We can now formally state the problem of decision tree construction.

Decision tree classification problem: Given a dataset $D = \{t_1, \dots, t_n\}$ where the t_i are independent random samples from a probability distribution P, find a decision tree classifier T such that the misclassification rate $R_T(P)$ is minimal.

A classification tree is usually constructed in two phases. In phase one, the *growth phase*, an overly large decision tree is constructed from the training data. In phase two, the *pruning phase*, the final size of the tree T is determined with the goal to minimize R_T. In this research, we concentrate on the tree growth phase, since due to its data-intensive nature it is a very time-consuming part of decision tree construction [MAR96, SAM96, GRG98]. (Cross-validation [BFOS84], a popular

```
Input: node n, partition D, split selection method CL
Output: decision tree for D rooted at n
TDTree(Node n, partition D, split selection method CL)
Apply CL to D to find the splitting criterion for n
if (n splits)
    Use best split to partition D into D1, D2
    Create children n1 and n2 of n
    TDTree(n1, D1, CL)
    TDTree(n2, D2, CL)
endif
```

Figure 1: Top-Down Tree Induction Schema

pruning technique for very small training datasets, requires construction of several trees from large subsets the data. Even though MDL-based pruning methods are more popular for large datasets [MAR96, RS98], our techniques can be used to speed up cross-validation for large training datasets as well.) How the tree is pruned is an orthogonal issue.

All decision 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 decision tree induction; for example, a specific instance of this schema for binary splits is shown in [MAR96]). This schema is depicted in Figure 1. All decision tree construction algorithms that we are aware of proceed according to this schema. Note that this schema requires one pass over the training database per level of the tree; the splitting criterion at a node n can not be computed unless the splitting criteria of all its ancestors in the tree are known.

2.2 Split Selection Methods

In this work, we consider split selection methods that produce binary splits. We will concentrate on *impurity-based* split selection methods for two reasons. First, this class of split selection methods is widely used and very popular [BFOS84, Qui86]; studies have shown that this class of split selection methods produces trees with high predictive accuracy [LLS97]. Second, most previous work in the database literature uses this class of split selection methods [MAR96, SAM96, FMM96, MFM+98, RS98], thus we can study the performance impact of our techniques while generating exactly the same output decision tree as previous methods. We would like to emphasize though, that our techniques described in Section 3 can be instantiated with other, not impurity-based split selection methods from the literature, e.g., QUEST [LS97], resulting in scalable decision tree construction algorithms. In Section 5, we also show experimental results with a non-impurity-based split selection method.

Impurity-based split selection methods calculate the splitting criterion by minimizing a concave *impurity function* $\mathtt{imp}_\theta$ such as the entropy [Qui86], the `gini`-index [BFOS84] or the index of correlation [MFM+98]. (Arguments for the concavity of the impurity function can be found in Breiman et al. [BFOS84].) The most popular split selection methods such as CART [BFOS84] and C4.5 [Qui86] fall into this group; previous work in the database literature generated scalable instantiations of these split selection methods [MAR96, SAM96, RS98, GRG98]. At each node, all predictor attributes X are examined and the impurity of the best split on X is calculated. The final split is chosen such that the value of $\mathtt{imp}_\theta$ is minimized. In the next section we briefly detail how $\mathtt{imp}_\theta$ is actually calculated for numerical predictor attributes. Calculation of $\mathtt{imp}_\theta$ for categorical predictor attributes is shown in the full paper.

2.2.1 Numerical Predictor Variables

In this section, we briefly explain how the class of impurity-based split selection methods computes the split point for numerical predictor variables.

Let X be a numerical predictor variable, i.e., splits on X will be of the form $X \leq x$ where $x \in \mathtt{dom}(X)$. At a node n, each potential split point $x \in \mathtt{dom}(X)$ induces a set of arguments $\vec{\theta_x} = \langle \theta^L_{n,X,x,1}, \ldots, \theta^L_{n,X,x,k}, \theta^R_{n,X,x,1}, \ldots, \theta^R_{n,X,x,k} \rangle$ for the impurity function $\mathtt{imp}_\theta$, where for $i \in \mathtt{dom}(C)$: $\theta^L_{n,X,x,i} \stackrel{\text{def}}{=} P(X \leq x, C = i | f_n)$, and $\theta^R_{n,X,x,i} \stackrel{\text{def}}{=} P(X > x, C = i | f_n)$. Informally, $\theta^L_{n,X,x,i}$ is the probability that for a randomly drawn tuple t that belongs to the family of node n, t's value for predictor attribute X is less than or equal to x and that t's class label has value i. Since x induces this set of arguments for $\mathtt{imp}_\theta$, we define $\mathtt{imp}_X(n, X, x) \stackrel{\text{def}}{=} \mathtt{imp}_\theta(\theta^L_{n,X,x,1}, \theta^L_{n,X,x,2}, \ldots, \theta^L_{n,X,x,k}, \theta^R_{n,X,x,1}, \ldots, \theta^R_{n,X,x,k})$. Since the underlying probability distribution P is not known, $\theta^L_{n,X,x,i}$ (respectively, $\theta^R_{n,X,x,i}$) is estimated from the training database D by $\widehat{\theta}^L_{n,X,x,i}$ (respectively, $\widehat{\theta}^R_{n,X,x,i}$), and $\mathtt{imp}_X$ is estimated from D by $\widehat{\mathtt{imp}}_X(n, X, x)$ as follows: $\widehat{\theta}^L_{n,X,x,i} \stackrel{\text{def}}{=} \frac{|\{t \in F_n : t.X \leq x \wedge t.C = i\}|}{|F_n|}$, $\widehat{\theta}^R_{n,X,x,i} \stackrel{\text{def}}{=} \frac{|\{t \in F_n : t.X > x \wedge t.C = i\}|}{|F_n|}$, and $\widehat{\mathtt{imp}}_X(n, X, x) \stackrel{\text{def}}{=}$ $\mathtt{imp}_\theta(\widehat{\theta}^L_{n,X,x,1}, \widehat{\theta}^L_{n,X,x,2}, \ldots, \widehat{\theta}^L_{n,X,x,k}, \widehat{\theta}^R_{n,X,x,1}, \ldots, \widehat{\theta}^R_{n,X,x,k})$. Informally, for node n and predictor attribute X, $\widehat{\theta}^L_{n,X,x,i}$ is the proportion of tuples in F_n with $t.X \leq x$ and class label i; $\widehat{\theta}^R_{n,X,x,i}$ is the proportion of tuples in F_n with $t.X > x$ and class label i.

3 BOAT—Bootstrapped Optimistic Decision Tree Construction

In this section, we present BOAT, a scalable algorithm for decision tree construction. Like the RainForest framework or the PUBLIC pruning strategy [GRG98, RS98], BOAT is applicable to a wide range of split selection methods. To our knowledge, BOAT is the first decision tree construction algorithm that constructs several levels of the tree in a single scan over the database. All previous algorithms that we are aware of make one scan over the database for each level of the tree. We will explain the algorithm in several steps. We first give an informal overview without technical details to give the intuition behind our approach (Section 3.1). We then describe our algorithm in detail in Sections 3.2 to 3.5. We assume in the remainder of this section that $\mathcal{CL}$ is an impurity-based split selection method that produces trees with binary splits.

3.1 Overview

Let T be the final tree constructed using split selection method $\mathcal{CL}$ on training database D. D does not fit in-memory, so we obtain a large sample $D' \subset D$ such that D' fits in-memory. We can now use a traditional main-memory decision tree construction algorithm to compute a *sample* tree T' from D'. Each node $n \in T'$ has a *sample splitting criterion* consisting of a *sample splitting attribute* and a *sample split point* (or a *sample splitting subset*, for categorical attributes). Intuitively, T' will be quite "similar" to the final tree T constructed from the complete training database D. But how similar is T' to T? Can we use the knowledge about T' to help or guide us in the construction of T, our final goal? Ideally, we would like to say that the final splitting criteria of T are very "close" to the sample splitting criteria of T'. But without any qualification and quantification of "similarity" or "closeness", information about T' is useless in the construction of T.

Consider a node n in the sample tree T' with numerical sample splitting attribute X_n and sample splitting predicate $X_n \le x$. By T' being close to T we mean that the final splitting attribute at node n is X and that the final split point is inside a confidence interval around x. If the splitting attribute at node n in the tree is categorical, we say that T' is close to T if the sample splitting criterion and the final splitting criterion are identical. Thus the notion of closeness for categorical attributes is more stringent because splitting attribute as well as splitting subset have to agree.

In Section 3.2, we show how a technique from the statistics literature called bootstrapping [ET93, AD97] can be applied to the in-memory sample D' to obtain a tree T' that is close (in the sense just mentioned) to T with high probability. Thus, using bootstrapping, we leverage the in-memory sample D' by extracting "more" information from it. In addition to a sample tree T', we also obtain confidence intervals that contain the final split points for nodes with numerical splitting attributes, and the complete final splitting criterion for nodes with categorical splitting attributes. We call the information at a node n obtained through bootstrapping the *coarse splitting criterion* at node n. This part of the algorithm, which we call the *sampling phase*, is described in Section 3.2.

Let us assume for the moment that the information obtained through bootstrapping is always correct. Using the sample $D' \subset D$ and the bootstrapping method, the coarse splitting criteria for all nodes n of the tree have been calculated and are correct. Now the search space of possible splitting criteria at each node of the tree is greatly reduced. We know the splitting attribute at each node n of the tree, for numerical splitting attributes we know a confidence interval of attribute values that contains the final split point, for categorical splitting attributes we know the final splitting subset exactly. Consider a node n of the tree with numerical splitting attribute. To decide on the final split point, we need to examine the value of the impurity function only at the attribute values inside the confidence interval. If we had all the tuples that fall inside the confidence interval of n in-memory, then we could calculate the final split point exactly by calculating the value of the impurity function at all possible split points inside the confidence interval. To bring these tuples in-memory, we make one scan over D and keep all tuples that fall inside a confidence interval at any node in-memory. Then we postprocess each node with a numerical splitting attribute to find the exact value of the split point using the tuples collected during the database scan. This postprocessing phase of the algorithm, the *cleanup phase*, is described in Section 3.3.

Attrib. Type	Coarse Splitting Criterion
Categorical	The splitting attribute X_n $Y \subset \mathtt{dom}(X_n)$ such that $\mathtt{imp}_X(n, X_n, Y)$ is minimized
Numerical	The splitting attribute X_n An interval $[i_n^L, i_n^R]$ that contains the final split point

Figure 2: Coarse splitting critera

The coarse splitting criterion at a node n obtained from the sample D' through bootstrapping is only correct with high probability. This means that occasionally the final splitting attribute is different from the sample splitting attribute, or, if the sample splitting attribute is equal to the final splitting attribute, it could happen that the final split point is outside the confidence interval, or that the final splitting subset is different from the sample splitting subset. Since we want to guarantee that our method generates exactly the same tree as if the complete training dataset were used, we have to be able to check whether the coarse splitting criteria actually are correct. In Section 3.4, we present a necessary condition that signals whenever the coarse splitting criterion is incorrect. Thus, whenever the coarse splitting criterion at a node n is not correct, we will detect it during the cleanup phase and can take necessary measures as explained in Section 3.5. Therefore we can guarantee that our method *always* finds *exactly the same tree* as if a traditional main-memory algorithm were run on the complete training dataset.

3.2 Coarse Splitting Criteria

We begin by formally defining the statistics that we assume to exist at each node n at the beginning of the cleanup phase. We call these statistics the *coarse splitting criterion* at node n. Then, we show how to calculate a coarse splitting criterion that is correct with high probability at each node n.

Informally, the coarse splitting criterion at n restricts the set of possible splitting criteria at n to a small set; it is a "coarse view" of the final splitting criterion. The coarse splitting criterion for the two attribute types is shown in Figure 2; its first part is called the *coarse splitting attribute*. At a node n with numerical splitting attribute X_n, the second part of the coarse splitting criterion consists of an interval of attribute values such that the final split point on X_n is inside the interval with high probability. Formally, assume that the final splitting predicate at node n is $X_n \le x_n^*$. Then the second part of the coarse splitting criterion at n consist of an interval $[i_n^L, i_n^R]$, $i_n^L, i_n^R \in \mathtt{dom}(X_n)$ such that $i_n^L \le i_n^R$, and

$x_n^* \in [i_n^L, i_n^R]$. Thus, to find the final split point, the value of the impurity function needs to be examined only at attribute values $x \in [i_n^L, i_n^R]$.

We now address the computation of the coarse splitting criterion. The main idea is to take a large sample D' from the training database D and use an in-memory algorithm to construct a sample tree T'. Then we use a technique from the statistics literature called bootstrapping [ET93, AD97] that allows us to quantify, at each node n, how accurate the splitting criterion obtained from the sample is with respect to the final splitting criterion. Recall that for a node $n \in T'$, its splitting criterion is called *sample splitting criterion* to emphasize that it has been computed from the in-memory sample $D' \subset D$; the final splitting criterion using the complete training database D might be different. We will also use the terms *sample splitting attribute, sample split point* and *sample splitting subset*.

We use bootstrapping to obtain the coarse splitting criterion as follows. First, we construct b bootstrap trees $T_1, \dots, T_b$, constructed from training samples $D_1, \dots, D_b$ obtained by sampling with replacement from D'. Then we process the trees top-down. For each node n, we check whether the b bootstrap splitting attributes at n are identical; if not, then we delete n and its subtree in all bootstrap trees. If at a node n the b bootstrap splitting attributes are the same categorical splitting attribute X_n, we check whether all bootstrap splitting subsets are identical. If not, then we also delete n and its subtree in all bootstrap trees. The intuition for such stringent treatment of categorical splitting attributes is that as soon as two subtrees split on different subsets, the subtrees are incomparable. Then for each node n in the remaining tree, we set the coarse splitting attribute to be the bootstrap splitting attribute. If the bootstrap splitting attribute at node n is categorical, we set the coarse splitting subset equal to the bootstrap splitting subset. If the bootstrap splitting attribute at node n is numerical, we have b bootstrap split points from which we can obtain a confidence interval $[i_n^L, i_n^R]$ for the final split point, such that with high probability the final split point $x_n^* \in [i_n^L, i_n^R]$. The level of confidence can be controlled by increasing the number of bootstrap repetitions.

3.3 From Coarse to Exact Splitting Criteria

In this section, we describe part of the cleanup phase of the algorithm. We show an algorithm that takes as input the sample tree T' and the coarse splitting criteria. The algorithm makes one scan over the training database while collecting a small subset of the training database in-memory. (We will assume without loss of generality that this subset of tuples fits in-memory; the implementation used in the experimental evaluation in Section 5 writes temporary files to disk to be truly scalable. We further discuss this issue in Section 3.5.) Assuming that the coarse splitting criteria are correct, the in-memory information is used to compute the final splitting criterion at n. Thus, assuming we are given the coarse splitting criterion at each node n and the coarse splitting criteria are actually correct, we can compute the final tree in only one scan over the training database. In the remainder of this section, we assume that the coarse splitting criteria are correct; we show how this assumption can be checked in the next section.

In order to describe the algorithm precisely, let us introduce the following notation. Let S be a set of tuples. At node n with numerical splitting attribute X_n and confidence interval $[i_n^L, i_n^R]$ from the coarse splitting criterion, define $l_n(S) \stackrel{\text{def}}{=} \{t : t \in S : t.X_n \leq i_n^L\}$, $i_n(S) \stackrel{\text{def}}{=} \{t : t \in S : t.X_n > i_n^L \wedge t.X_n \leq i_n^R\}$, and $r_n(S) \stackrel{\text{def}}{=} \{t : t \in S : t.X_n > i_n^R\}$.

Given the coarse splitting criterion at a node n, how can the final splitting criterion be computed? If the splitting attribute at n is categorical, the coarse splitting criterion is equal to the final splitting criterion; no extra computation is necessary. Thus, let us concentrate on the case where the splitting attribute at n is numerical. Since the final split point x_n^* is inside the confidence interval $x_n^* \in [i_n^L, i_n^R]$, we have to examine all possible split points inside the interval. Assume that n is the root node of the tree. To estimate the quality of an attribute value $x \in [i_n^L, i_n^R]$ as potential split point, the values $\widehat{\theta}^L_{n,X_n,x,i}$ and $\widehat{\theta}^R_{n,X_n,x,i}$ are needed as arguments to the impurity function $\widehat{\mathtt{imp}_X}(n, X_n, x)$ in order to calculate the impurity at x (see Section 2.2). Assume that we make one scan over the training database, during which we compute $\widehat{\theta}^L_{n,X_n,i_n^L,i}$ and $\widehat{\theta}^R_{n,X_n,i_n^R,i}$ for all $i \in \mathtt{dom}(C)$. In addition, we keep all tuples $t \in i_n(F_n)$ in-memory. Then after the scan, the arguments to the impurity function $\widehat{\mathtt{imp}_X}(n, X_n, x)$ for each $x \in [i_n^L, i_n^R]$ can be calculated as follows:

$$\widehat{\theta}^L_{n,X_n,x,i} = \frac{|\{t \in F_n : t.X \leq x \wedge\ t.C = i\}|}{|F_n|}$$
$$= \frac{\widehat{\theta}^L_{n,X_n,i_n^L,i} \cdot |F_n| + |\{t \in i_n(F_n) : t.X \leq x \wedge\ t.C = i\}|}{|F_n|},$$

the value $\widehat{\theta}^R_{n,X_n,x,i}$ is calculated similarly. Thus, in one scan over the training database, we can find the final splitting criterion at the root node n given the coarse splitting criterion at n: we retain the set $i_n(F_n)$ in-memory and then calculate the value of the impurity function $\widehat{\mathtt{imp}_X}(n, X_n, x)$ at each potential split point $x \in [i_n^L, i_n^R]$ using the tuples in-memory.

Consider now the computation of the final split point $x_{n'}^* \in [i_{n'}^L, i_{n'}^R]$ for the left child n' of the root node n. After having computed the splitting criterion at n, we can use the same algorithm for n': Make one pass over D, while collecting $i_{n'}(F_{n'})$ in-memory and calculating $|F_{n'}|$, $\widehat{\theta}^L_{n',X_{n'},i_n^L,i}$ and $\widehat{\theta}^R_{n',X_{n'},i_n^R,i}$ for $i \in \mathtt{dom}(C)$. This method will result in an algorithm that makes one scan over D per node in T during the cleanup phase. Is it possible to collect $i_{n'}(F_{n'})$ and $\widehat{\theta}^L_{n',X_{n'},i_n^L,i}$ and $\widehat{\theta}^R_{n',X_{n'},i_n^R,i}$ for $i \in \mathtt{dom}(C)$ during the first scan? Unfortunately, the answer is no. If for a tuple $t \in D$, $t \in l_n(D)$, then t belongs to $F_{n'}$. Consider a tuple $t \in i_n(F_n) \cap i_{n'}(F_{n'})$. During the scan of D, the final splitting criterion of node n is not known yet. Thus, for all tuples $t \in i_n(F_n)$, we cannot decide yet whether to send t

to the left child or the right child of n. Therefore after the first scan, $F_{n'}$ is not complete yet, because some tuples got "stuck" at node n. Thus, if at a node n, $t \in i_n(F_n)$, we retain t in a set of tuples S_n and stop. If $t \notin i_n(F_n)$, then we update $\widehat{\theta}^L_{n,X_n,i^L_n,i}$ or $\widehat{\theta}^R_{n,X_n,i^R_n,i}$ and recursively process t by sending it to its subtree. The result of the scan is a set of tuples S_n at each node; for the root node n, $S_n = i_n(F_n)$, for a non-root node n, $S_n \subset i_n(F_n)$. Note that after the scan at a node n, if a tuple $t \in i_n(F_n) \setminus S_n$, then $t \in i_{n'}(F_{n'})$ for some ancestor n' of n. Thus, for each node n, all the tuples $t \in i_n(F_n)$ are actually in-memory, either in S_n or in $S_{n'}$ for some ancestor n' of n. This observation allows for the following top-down processing of the tree after the scan: We start at the root node n and find its splitting criterion. Then we process all tuples $t \in S_n$ recursively by the tree as during the scan of D. Since the splitting criterion at n has been computed we know the correct subtree for each tuple. Finally, we recurse on n's children. Whenever a node n is processed, $S_n = i_n(F_n)$, because all the tuples $t \in i_{n'}(F_{n'})$ for all ancestors n' of n have been processed and distributed to their respective children.

Summarizing the results from this section, we have shown how to obtain the final splitting criterion for all nodes of the tree simultaneously in only one scan over the training database D, given that we know the coarse splitting criterion at each node n of the tree.

3.4 How To Detect Failure

In Section 3.2, we showed how to compute a coarse splitting criterion that is correct with high probability. In order to make the algorithm deterministic, we need to check during the cleanup phase whether the first part of the coarse splitting criterion, the coarse splitting attribute, is actually the final splitting attribute. In addition, given that this premise holds, we have to check that the second part of the coarse splitting criterion is correct. If not, then for a categorical splitting attribute, the split might involve a different subset. For a numerical splitting attribute, the split might be outside the confidence interval.

Let us first address the case of how to detect whether the second part of the coarse splitting criterion is correct, assuming that the first part is correct. That is, for now we assume that the coarse splitting attribute X is equal to the final splitting attribute and concentrate on checking the second part of the coarse splitting criterion. The second part of the coarse splitting criterion for a categorical splitting attribute X consists of the coarse splitting subset Y'; we have to check whether Y' is equal to the final splitting subset Y. We perform this check by constructing the values $\theta_{n,X,\{x\},i}$ for each $x \in \mathtt{dom}(X)$ and $i \in \mathtt{dom}(C)$ during the cleanup scan in-memory.

The second part of the coarse splitting criterion for a numerical splitting attribute consists of a confidence interval $[i^L_n, i^R_n]$. Let $x'_n \in [i^L_n, i^R_n]$ be the attribute value with the minimum value of the impurity function, Let $i' \stackrel{\text{def}}{=} \widehat{\mathtt{imp}}_X(n, X, x'_n)$. In order to be sure that the final split point is not outside the confidence interval, we have to check that i' is the global minimum of the impurity function, and not just the local minimum inside the confidence interval. Conceptually, we have to calculate the value of the impurity function at every $x \in \mathtt{dom}(X)$, $x \notin [i^L_n, i^R_n]$ and compare it with i'. For this calculation, we need to construct all values $\theta^L_{n,X,x,i}$ and $\theta^R_{n,X,x,i}$ during the cleanup scan in-memory. But since we construct several levels of the tree together, it is prohibitive to keep all these values simultaneously in-memory during the cleanup-scan. (Constructing these values in-memory is analogous to constructing the AVC-sets [GRG98] of predictor attribute X for all nodes concurrently in main memory.) So we need a method that allows us to conclude that i' is the global minimum of the impurity function over all attribute values of X without constructing *all* values $\theta^L_{n,X,x,i}$ and $\theta^R_{n,X,x,i}$ in-memory. The remainder of this section addresses this issue.

Consider node n with numerical predictor attribute X. (We will drop the dependencies on n and X from the notation in the following discussion.) Let N^i be the count of tuples in F_n with class label i. Let $x \in \mathtt{dom}(X)$ be an attribute value and let $n^i_x \stackrel{\text{def}}{=} |\{t \in F_n : t.X \leq x \wedge t.C = i\}|$ for $i \in \mathtt{dom}(C)$. Thus each attribute value $x \in \mathtt{dom}(X)$ uniquely determines a tuple of values $(n^1_x, \ldots n^k_x)$, called the *stamp point* of x [FMMT96a, FMMT96b, FMM96, MFM+98]. Thus, F_n induces a set of stamp points in the $k-$dimensional plane. At a node n, let N^i be the number of tuples in the family of n with class label i; formally, $N^i \stackrel{\text{def}}{=} |t \in D : t \in F_n \wedge t.C = i\}$. Since at node n, N^i is fixed for all $i \in \mathtt{dom}(C)$, the tuple $(n^1_x, \ldots, n^i_x)$ uniquely determines the value of the impurity function $\widehat{\mathtt{imp}}_X$ at attribute value x because we can rewrite the arguments to the impurity function as follows:

$$\widehat{\theta}^L_{n,X,x,i} = \frac{n^i_x}{|F_n|}, \text{ and } \widehat{\theta}^R_{n,X,x,i} = \frac{N^i - n^i_x}{|F_n|}$$

We can define a new function $\mathtt{imp}_S$ on the stamp points as follows: $\widehat{\mathtt{imp}}_S(n_1, \ldots, n_k) \stackrel{\text{def}}{=}$

$$\mathtt{imp}_\theta(\frac{n_1}{|F_n|}, \ldots, \frac{n_k}{|F_n|}, \frac{N^1 - n_1}{|F_n|}, \ldots, \frac{N^k - n_k}{|F_n|})$$

Let $x \in \mathtt{dom}(X)$ and let $(n^1_x, \ldots n^k_x)$ be the stamp point of attribute value x at node n. By construction of $\widehat{\mathtt{imp}}_S$, it holds that for $x \in \mathtt{dom}(X) : \widehat{\mathtt{imp}}_S(n^1_x, \ldots n^k_x) = \mathtt{imp}_X(n, X, x)$. Consider two attribute values $x_1 < x_2$ and let $P_{x_1,x_2} \stackrel{\text{def}}{=} \{(n^1_{t.X}, \ldots, n^k_{t.X}) : t \in F_n \wedge t.X \geq x_1 \wedge t.X \leq x_2\}$, i.e., P_{x_1,x_2} is the set of stamp points of all attribute values between x_1 and x_2 that occur in F_n. Note that if $x_1 > x_2$, $n^i_{x_1} \geq n^i_{x_2}$ for all $i \in \mathtt{dom}(C)$. Since $\mathtt{imp}_X$ and thus $\mathtt{imp}_S$ are concave, the minimum value of the impurity function $\widehat{\mathtt{imp}}_X$ will be on the convex hull of the stamp points P_{x_1,x_2} [Man94, FMMT96a, FMMT96b, FMM96, MFM+98]. Because the convex hull is enclosed in the hyper-rectangle defined by the 2 corner points $(n^1_{x_1}, \ldots, n^k_{x_1})$ and

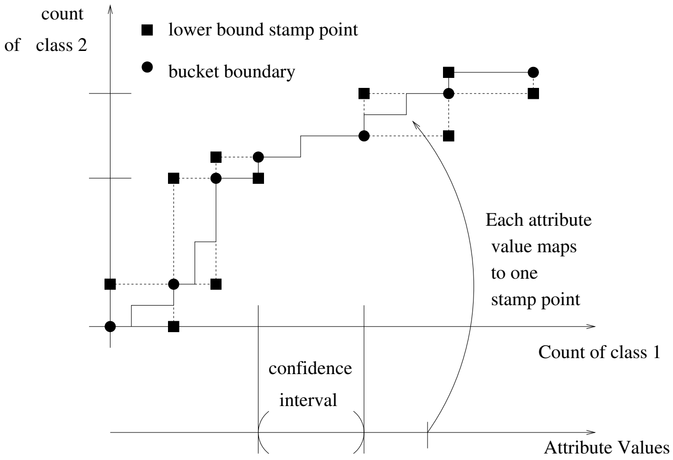

Figure 3: Mapping from attribute values to stamp points and the lower bound

$(n^1_{x_2}, \ldots, n^k_{x_2})$, it is enclosed by the 2^k corner points. For example, $(n^1_{x_1}, n^2_{x_2}, n^3_{x_1}, \ldots, n^k_{x_1})$ is one of the corner points. Thus, the value of the impurity function for all attribute values inside the interval $[x_1, x_2]$ can be lower-bounded by the value of the impurity function at the 2^k corner points. For example, if $k = 2$, the four corner points are $(n^1_{x_1}, n^2_{x_1})$, $(n^1_{x_2}, n^2_{x_2})$, $(n^1_{x_1}, n^2_{x_2})$, and $(n^1_{x_2}, n^2_{x_1})$. Figure 3 illustrates this situation for the case of two class labels. In the general case with k class labels, the following lemma holds.

Lemma 3.1 Let n be a node in the tree, X be a numerical predictor attribute and $x_1 < x_2, x_i \in \mathtt{dom}(X)$ be two attribute values with stamp points $(n^1_{x_1}, \ldots, n^k_{x_1})$ and $(n^1_{x_2}, \ldots, n^k_{x_2})$, respectively. Let P_{x_1,x_2} be the set of stamp points of all attribute values between x_1 and x_2 that occur in F_n. Let $\mathtt{imp}_S$ be a concave impurity function. Let S be the set of 2^k corner points of the hyper-rectangle defined by $(n^1_{x_1}, \ldots, n^k_{x_k})$ and $(n^1_{x_2}, \ldots, n^k_{x_2})$: $S \stackrel{\text{def}}{=} \{(n^1_{x_1}, \ldots, n^k_{x_1}), (n^1_{x_1}, n^2_{x_2}, \ldots, n^k_{x_k}), \ldots, (n^1_{x_2}, \ldots, n^k_{x_2})\}$. Then

$$\min_{(n_1,\ldots,n_k)\in P_{x_1,x_2}} \mathtt{imp}_S(n_1, \ldots, n_k) \quad \geq \quad \min_{(n_1,\ldots,n_k)\in S} \mathtt{imp}_S(n_1, \ldots, n_k)$$

Proof: This is an application of a result in Mangasarian [Man94] to the decision tree setting. □

Before we discuss how we use Lemma 3.1, we define the notion of a discretization f of a numerical variable X. For a random variable X with a numerical domain $\mathtt{dom}(X)$, we call a function $f : \mathtt{dom}(X) \mapsto \mathbf{N}$ a discretization of X if $x_i < x_j$ implies that $f(x_i) \leq f(x_j)$ for $x_i, x_j \in \mathtt{dom}(X)$. (Actually, $f(X)$ is a new random variable with domain $\mathbf{N}$, where $\mathbf{N}$ denotes the set of natural numbers.) We call each $k \in \mathbf{N}$ a *bucket* of f, and if $f(x) = k$ we say that x *belongs to bucket* k (under f). We call a value $x' \in \mathtt{dom}(X)$ such that $\forall x \in \mathtt{dom}(X) : (x < x' \Rightarrow f(x) \leq f(x')) \wedge (x > x' \Rightarrow f(x') > f(x))$ a *bucket boundary*.

At each node n, we calculate a discretization f for the splitting attribute from the sample D'. During the cleanup phase, we construct the stamp points at the bucket boundaries of f through simple counting. We use Lemma 3.1 to calculate a lower bound on the value of the impurity function for each bucket; let i be the minimum of all these lower bounds. Then we compare i with the minimum value of the impurity function i' calculated during the cleanup phase for the splitting attribute. If $i < i'$, then the final split point might fall into bucket B instead of inside the confidence interval; in this case we discard node n and its subtree. Lemma 3.1 therefore gives us a condition that is necessarily true whenever the final split point is outside the confidence interval. Thus, in the case of a numerical splitting attribute, we can detect whether the second part of the coarse splitting criterion is correct.

It remains to show how we can detect the case that the first part of the coarse splitting criterion, namely the choice of splitting attribute, is incorrect. Consider a node n and let the minimum value of the impurity function given the coarse splitting criterion is true be i', i.e., if the coarse splitting attribute X_n is categorical, i' is the value of the impurity function of the coarse splitting subset, if the coarse splitting attribute X_n is numerical, i' is the minimum value of the impurity function over all attribute values inside the confidence interval. Whenever n is processed during the cleanup phase, we can calculate the minimum value of the impurity function for all categorical attributes exactly and compare it with i'. For the remaining numerical variables, we calculate discretizations during the sampling phase and obtain the values of the stamp points of the discretization boundaries during the cleanup phase. Then we use Lemma 3.1 to lower bound the value of the impurity function at all buckets. If i' is still the global minimum, then the splitting attribute from the coarse splitting criterion is actually the final splitting attribute.

How do we find a "good" discretization f for a numerical predictor attribute X at node n in the tree? Note that the only purpose f serves is to allow the application of Lemma 3.1 to (1) check whether the actual split point is inside the confidence interval in case X is the coarse splitting attribute or to (2) check whether the final splitting attribute could be X, in case X is not the coarse splitting attribute. If f has too few buckets, the lower bound produced by Lemma 3.1 will be very crude; thus the lemma might too often indicate that a bucket could have a split point with a lower value $i < i'$ of the impurity function, even though there is no attribute value in the discretization bucket that actually achieves i. Too many buckets of f are not a problem; the lower bound of Lemma 3.1 will be very tight. But since BOAT requires discretizations for all numerical predictor attributes at each node of the subtree currently under construction, we cannot afford to have overall too many discretization buckets due to main memory constraints. What we would like is to construct at each node as many buckets as "necessary". How many buckets are necessary at node n for numerical predictor attribute X? We construct the bucket boundaries before the cleanup scan from the in-memory sample as follows. We scan the attribute values occurring in F'_n constructed from the sample D' from smallest to largest value. If the lower bound of the current bucket is much higher from the estimated lowest value of the impurity function at node n, then the bucket can be enlarged. Otherwise a new bucket boundary is set. This procedure constructs many buckets in regions of the attribute space where the value of the impurity function is close to the overall minimum and the bounds produced by Lemma 3.1 need to be quite tight in order not to signal a false alarm. The procedure constructs few buckets in regions where the value of the impurity function is much larger than the overall minimum.

Since we can detect all cases of incorrectness of the coarse splitting criterion at a node n, we have shown the following lemma to be correct.

Lemma 3.2 Consider a node n of the final decision tree and let i' be the minimum value of the impurity function given that the coarse splitting criterion at n is correct. If i' is not the global minimum of the impurity function at node n, then our algorithm will detect this case.

3.5 Putting the Parts Together

We now explain how the parts of the algorithm described in Sections 3.2 to 3.4 are put together to arrive at a fast, scalable, deterministic algorithm for decision tree construction.

We first take a sample $D' \subset D$ from the training database and construct a sample tree with coarse splitting criteria at each node using bootstrapping. Then we make a scan over the database D and process each tuple by "streaming it" down the tree. At the root node n, we first update the category-class-label counts for all categorical predictor attributes. Then we update the counts of the buckets for each numerical predictor attribute. If the splitting attribute from the coarse splitting criterion at n is categorical, we send t to the child node of n as predicted by the splitting criterion. If the splitting attribute from the coarse splitting criterion at n is numerical and t falls inside the confidence interval, we write t to a temporary file S_n at node n. Otherwise we send t down the tree. Note that we can stop tree construction if the size $|F_n|$ of the family of a node n is small enough to fit in-memory because in this case it is always cheaper to run a main-memory algorithm on F_n.

After the database scan, the tree is processed top-down. At each node, we use our lower bounding technique to check whether the global minimum value of the impurity function could be lower than i', the minimum impurity value calculated from either the complete information about the categorical splitting attribute or from examining the tuples in S_n. If the check is successful (i.e., i' is the global minimum), we are done with node n. If the check indicates that the global minimum could be less than i', we discard n and its subtree during the current construction and call our algorithm recursively on n after processing the rest of the tree. In most cases, the family of tuples F_n is already so small that F_n completely fits in-memory, and thus an in-memory algorithm can be used to finish construction of the subtree rooted at n.

4 Extensions to a Dynamic Environment

We outline briefly how the information about the coarse splitting criterion can be used to extend BOAT to support incremental updates of the decision tree in a dynamic environment where the training dataset changes over time through both insertions and deletions.

Consider the root node n of the tree. The training dataset D was generated by an unknown underlying probability distribution P. Since D is a random sample from P, all statistics obtained from D are only approximations of true parameters of the distribution. Now consider a new "chunk" of training data D_1 that needs to be incorporated into the tree. If D_1 is drawn from the same underlying distribution, the new tree $T_{D \cup D_1}$ that models $D \cup D_1$ will not be very "different" from T_D. This fuzzy notion of difference is actually exactly the same notion as described in Section 3.2. In Section 3.2, we are given a sample D' from an underlying distribution (represented by D) and would like to know how different the tree $T_{D'}$ is from the tree T_D. The coarse splitting criteria in $T_{D'}$ actually capture the randomness of D' by allowing the split point for numerical splitting attributes to fluctuate inside the confidence interval. Thus, another view of the coarse splitting criterion is that it captures a set of possible final splitting criteria, all highly likely given the underlying probability distribution P as captured by D.

Using the same statistical notion of difference as discussed in the previous paragraph, and as represented by the coarse splitting criterion, our algorithm to update the tree in a dynamic environment works as follows. We keep the information that we collected during the cleanup phase from D at each node n of the tree. Thus, associated with each node n with numerical predictor attribute, is a file S_n that contains all tuples that fell inside the confidence interval $[i_n^L, i_n^R]$ during the scan over D. To incorporate a new set of tuples D_1 into the tree, we stream the tuples $t \in D_1$ down the tree as if they were part of D and we were making the scan over D_1 during the cleanup phase. Following the processing of D_1, we again process the tree top-down, exactly as in the cleanup phase. If D_1 is also a random sample from the same underlying probability distribution, then by construction of the coarse splitting criterion, the final splitting criterion at node n will be included in the set of splitting criteria captured by the coarse splitting criterion of n and thus we can calculate the final splitting criterion at n exactly—all this while scanning D_1 exactly once! Deletions can be handled in the same way. Assume that D_1 expired and is removed from the training dataset. Then we can process D_1 as for insertion, but with the difference that instead of inserting tuples, we remove the respective tuples from the tree and update the counts maintained to ensure detection of changes of the coarse splitting criterion.

This algorithm has the following properties. If D_1 is sufficiently different from D, then this will be detected by our lower-bound techniques which will indicate that the coarse splitting criterion at a node n is not correct any more. In this case, the affected part of the tree, namely the subtree rooted at n, needs to be rebuilt. Note that only the part of the tree in which the distribution has sufficiently changed needs to be rebuilt. This cost model is very attractive in a real-life setting: If new data arrives (or old data expires), but the changes in the training dataset are only due to random fluctuations (in the precise statistical sense), then the cost to update the tree is very low and involves only a scan over the dataset that is to be inserted or deleted. It there are changes in the distribution, the cost paid is proportional to the "seriousness" of the changes: if the splitting attribute at the root node changes, the whole tree needs to be completely rebuilt. But if the distribution changes only in a part of the attribute space, only the subtree that models that part of the space needs to be rebuilt. In addition, showing that statistically significant changes have happened in part of the tree is a valuable tool for the analyst who can be informed that specific parts of the tree have changed significantly, even though other parts of the tree might only have changed slightly inside the confidence intervals. This insight is much more than what could be extracted by just comparing the two trees T_D and $T_{D \cup D_1}$ (or T_D and $T_{D \setminus D_1}$). Using such a comparison, it is possible to point out changes in the splitting predicates but it is not possible to assess whether these changes are due just to the randomness in the overall process or due to a change in the underlying distribution.

We emphasize that, as in the static case, the adaptation of our algorithm to a dynamic environment always guarantees that the tree constructed is exactly the same tree as if a traditional algorithm was run on the changed training dataset.

5 Experimental Evaluation

The two main performance measures for classification tree construction algorithms are: (i) the predictive quality of the resulting tree, and (ii) the decision tree construction time [LLS97]. BOAT can be instantiated with any split selection method from the literature that produces binary trees *without modifying the result of the algorithm*. Thus, quality is an orthogonal issue to our algorithm, and we can concentrate solely on decision tree construction time. In the remainder of this section we show the results of a preliminary performance study of our algorithm for a variety of datasets for impurity-based split selection methods. Based on some observations about impurity-based split selection methods from our experiments, we also show performance results for another non-impurity based split selection method. The results demonstrate that BOAT achieves significant performance improvements (two to five times). Finally, we also show some performance results for classification tree maintenance in a dynamic environment.

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 benchmark datasets to evaluate scalability results. The largest dataset in the often used Statlog collection of training databases [MST94] contains only 57000 records, and the largest training dataset considered in [LLS97] has 4435 tuples. We therefore use the synthetic data generator introduced by Agrawal et al. in [AIS93], henceforth referred to as `Generator`. This data generator has been used previously in the database literature to study the performance of decision tree construction algorithms [SAM96, RS98, GRG98]. The synthetic data has nine predictor attributes. Each tuple generated by the synthetic data generator has a size of 40 bytes (assuming binary files). Included in the generator are classification functions that assign class labels to the records produced. We selected three of the functions (Function 1, 6 and 7) introduced in [AIS93] for our performance study. In Function 1, two predictor attributes carry predictive power with respect to the class label, Function 6 involves three predicates, and in Function 7 the class label depends on a linear combination of four predictor attributes [AIS93]. Note that our selection of predicates adheres to the methodology used in the Sprint, PUBLIC and RainForest performance studies [SAM96, RS98, GRG98].

We compare our algorithm to the RainForest algorithms, which were shown to outperform previous work [GRG98]. The feasibility of the RainForest family of algorithms requires a certain amount of main memory that depends on the size of the initial AVC-group [GRG98], whereas BOAT does not have any a-priori main memory requirements. Thus we compared BOAT to the two extremes in the RainForest family of algorithms. We chose the fastest algorithm, RF-Hybrid, requiring the largest amount of main memory, and the slowest algorithm, called RF-Vertical, requiring the smallest amount of main memory. Since we are interested in the behavior of our algorithm for datasets that are larger than main memory, we stopped tree construction for leaf nodes whose family would fit in-memory. Any smart implementation would switch to a main-memory tree construction at this point.

In all the experiments reported here, we took an initial sample of size 200000 tuples from the training database and then performed 20 bootstrap repetitions with a subsample size of 50000 tuples each. All our experiments were performed on a Pentium Pro with a 200 Mhz processor running Solaris X86 version 2.6 with 128 MB of main memory. All algorithms are written in C++ and were compiled using `gcc` version `pgcc`−2.90.29 with the `-O3` compilation option.

5.2 Scalability Results

First, we examined the performance of BOAT as the size of the input database increases. For Algorithms RF-Hybrid and RF-Vertical, we set the size of the AVC-group buffer to 3 million and 1.8 million entries, respectively. For this experiment, we stopped tree construction at 1.5 million tuples, which corresponds to a size of the family of tuples at a node of 60 MB. (The threshold is set to 60 MB, because RF-Hybrid uses around 60 MB of main memory in the optimized version that we compared BOAT with.) Figures 4 to 6 show the overall running times of the algorithms as the number of tuples in the training database increases from 2 million to 10

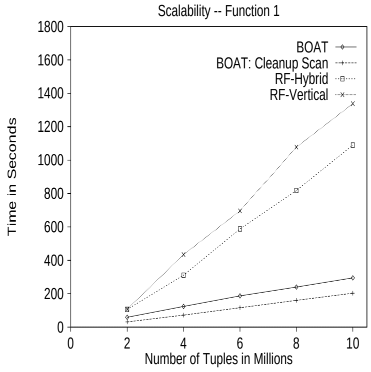

Figure 4: Overall Time: F1

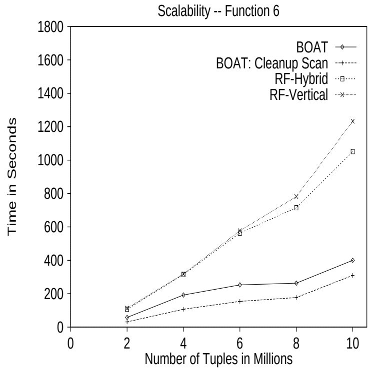

Figure 5: Overall Time: F6

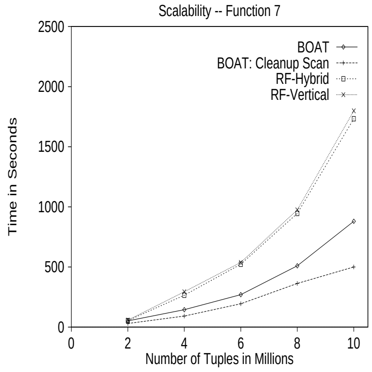

Figure 6: Overall Time: F7

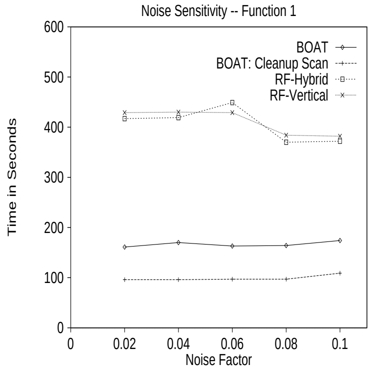

Figure 7: Noise: Time F1

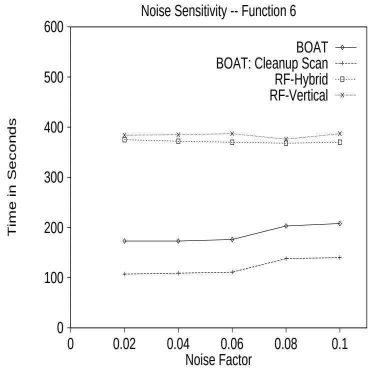

Figure 8: Noise: Time F6

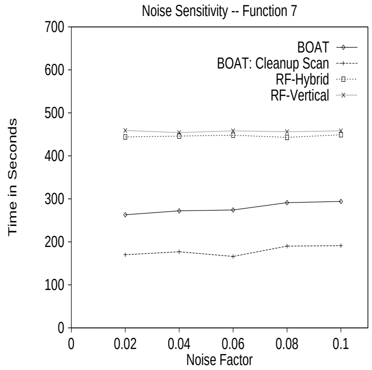

Figure 9: Noise: Time F7

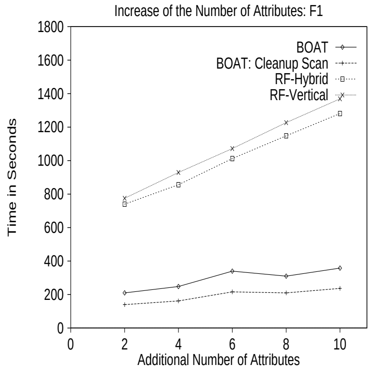

Figure 10: Extra Attributes: F1

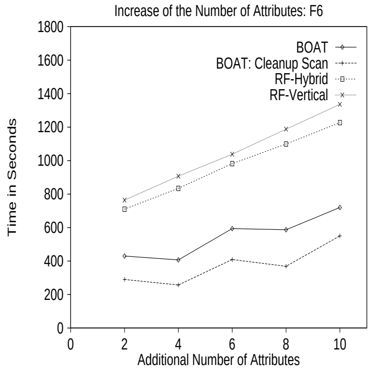

Figure 11: Extra Attributes: F6

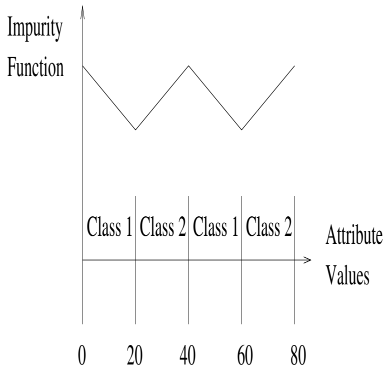

Figure 12: Instability

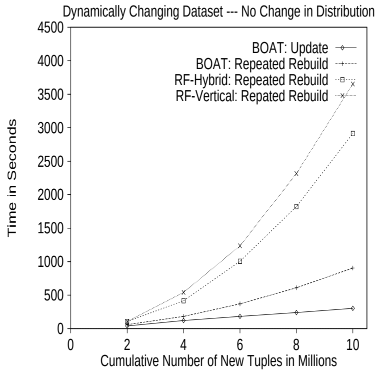

Figure 13: Dynamic: No Change

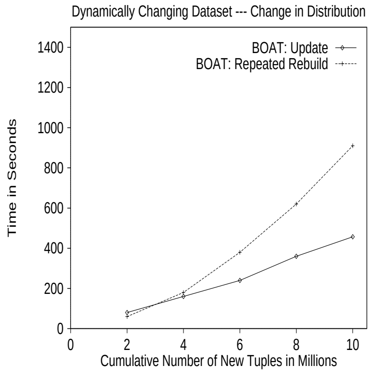

Figure 14: Dynamic: Change

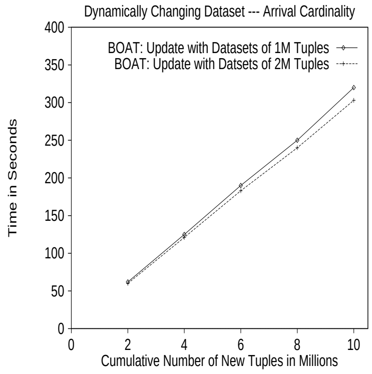

Figure 15: Dynamic: Small Updates

million tuples. The trees produced by Function 7 have more nodes before the threshold is reached, thus tree growth takes longer than for the other two functions. BOAT outperforms both RF-Hybrid and RF-Vertical in terms of running time; for Functions 1 and 6, BOAT is faster by a factor of three and for Function 7 by a factor of two. Since tree construction was stopped at 1.5 million tuples, BOAT achieves no speedup yet for training database sizes of 2 million tuples (the resulting tree has just three leaf nodes before the switch to the in-memory implementation occurs). But the speedup becomes more and more pronounced as the size of the training database increases.

We also examined the effect of noise on the performance of BOAT. We expected that noise would have a small impact on the running time of BOAT. Noise mainly affects splits at lower levels of the tree, where the relative importance between individual predictor attributes decreases, since the most important predictor attributes have already been used at the upper levels of the tree to partition the training dataset. Figures 7 to 9 compare the overall running times on datasets of size 5 million tuples while increasing the percentage of noise in the data from 2 to 10 percent. As in the previous experiment, we stopped tree construction at 1.5 million tuples. The figures show that the running time of BOAT is not dependent on the level of noise in the data.

Figure 10 shows the effect of adding extra attributes with random values to the records in the input database. (Due to space limitations we only show the Figure for Function 1, the behavior is similar for the other functions.) Adding attributes increases tree construction time since the additional attributes need to be processed, but does not change the final decision tree. (The split selection method will never choose such a "noisy" attribute as the splitting attribute.) BOAT exhibits a roughly linear scaleup with the number of additional attributes added.

During the experiments with BOAT we found out that the instability of impurity-based split selection methods deteriorates our performance results. By instability of a split selection we mean that minimal changes in the training dataset can result in selection of a very different split point. As an extreme example, consider the situation shown at node n in Figure 12; n is a numerical attribute with 81 attribute values (0 to 80). Assume that there is nearly the same number of tuples inside each interval of length 20 and assume that the final split is at attribute value 20. Through insertion or deletion of just a few tuples, the global minimum of the impurity function can be made to jump from attribute value 20 to attribute value 60 since both minima are very close to each other. Thus if bootstrapping is applied to the situation depicted in Figure 12, about half the time the split point will be very close to attribute value 20 and the remaining times the split point will be very close to attribute value 60. Since the two splits are so far apart, the subtrees grown from the two splits will very likely be different, and thus tree growth stops at node n since two bootstrap samples disagree on the splitting attributes of the children of n.

5.3 Performance Results for Dynamically Changing Datasets

Since BOAT allows to update the tree dynamically, we also compared the performance of the update operation in BOAT to a repeated re-build of the tree. Due to space constraints, we show performance numbers only for insertions of tuples into the training datasets; since insertion and deletion of tuples are handled symmetrically, the performance results for deletions are analogous.

In our first experiment, we examined the performance of the update operation for a changing training dataset whose underlying data distribution does not change. We ran BOAT on a dataset generated by Function 1 from the synthetic data generator. Then we generated chunks of data of size 2 million tuples each from the same underlying distribution, but we set the level of noise in the new data to 10%. Figure 13 shows the cumulative time taken to incorporate the new data into the tree. Note that the time taken for BOAT is independent of the size of the very first dataset that was used to construct the original tree. If the underlying data distribution does not change, the in-memory information about the coarse splitting criteria and the tuples inside the confidence intervals that BOAT maintains is sufficient to incorporate the new data and to update the tree without examining the complete original training database. To give a very conservative comparison of the update operation to repeated re-builds, we assumed the size of the original dataset to be zero. Thus the running time for the repeated re-builds shown in Figure 13 is the cumulative time needed to construct a tree on datasets of size 2 to 10 million tuples for the respective algorithms. Figure 15 shows a comparison of running times for arrival chunks of cardinality of size 1 million tuples versus 2 million tuples. The two curves are nearly identical.

What happens if the underlying distribution changes? In our next experiment we modified Function 1 from the synthetic data generator such that the tree in part of the attribute space is different from the original tree generated by Function 1. The results are shown in Figure 14. Even though in the incremental algorithm parts of the tree get rebuild, the incremental algorithm outperforms repeated rebuilds by a factor of 2.

6 Related Work

Agrawal et al. introduce an interval classifier that could use database indices to efficiently retrieve portions of the classified dataset using SQL queries [AGI+92]. Fukuda et al. construct decision trees with two-dimensional splitting criteria [FMM96]. The decision tree classifier SLIQ [MAR96] was designed for large databases but uses an in-memory data structure that grows linearly with the number of tuples in the training database. This limiting data structure was eliminated by Sprint, a scalable data access method, that removes all relationships between main memory and size of the dataset [SAM96]. In recent work, Morimoto et al. developed algorithms for decision tree construction for categorical predictor variables with large domains [MFM+98]. Rastogi and Shim developed PUBLIC, a MDL-based pruning algorithm for binary trees that is interleaved with the tree growth phase [RS98]. In the RainForest Framework, Gehrke et al. proposed a generic scalable data access method that can be instantiated with most split selection methods from the literature [GRG98], resulting in a scalable classification tree construction algorithm.

The tree induction algorithm ID5 restructures an existing tree in-memory [Utg89] in a dynamic environment under the assumption that the complete training database fits in-memory. Utgoff et al. extended this work and presented a series of restructuring operations that can be used to derive a decision tree construction algorithm for a dynamically changing training database [UBC97] while maintaining the optimal tree. But their techniques also assume that the training database fits in-memory. Efron and Tibshirani [ET93] and Davison and Hinkley [AD97] both are excellent introductions to the bootstrap. In recent work, Megiddo and Ramakrishnan used a form of bootstrapping to assess the statistical significance of a set of association rules [MS98].

7 Conclusions

We introduced a new scalable algorithm `BOAT` for constructing decision trees from large training databases. `BOAT` is faster than the best existing algorithms by a factor of three while constructing exactly the same decision tree, and can handle a wide range of splitting criteria. Beyond improving performance, `BOAT` enhances the functionality of existing scalable decision tree algorithms in two major ways. First, `BOAT` is the first scalable algorithm that can maintain a decision tree incrementally when the training data set changes dynamically. Second, `BOAT` greatly reduces the number of database scans, and offers the flexibility of computing the training database on demand instead of materializing it, as long as random samples from parts of the training database can be obtained. In addition to developing the `BOAT` algorithm and proving it correct, we have implemented it and presented a thorough performance evaluation that demonstrates its scalability, incremental processing of updates, and speed-up over existing algorithms.

Acknowledgements: We thank Anand Vidyashankar for introducing us to the bootstrap.

References

[AD97] A.C.Davison and D.V.Hinkley. *Bootstrap Methods and their Applications*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997.

[AGI+92] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An interval classifier for database mining applications. VLDB 1992.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

[BFOS84] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, 1984.

[ET93] B. Efron and R. J. Tibshirani. *An introduction to the bootstrap*. Chapman & Hall, 1993.

[FMM96] T. Fukuda, Y. Morimoto, and S. Morishita. Constructing efficient decision trees by using optimized numeric association rules. VLDB 1996.

[FMMT96a] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization. SIGMOD 1996.

[FMMT96b] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Mining optimized association rules for numeric attributes. PODS 1996.

[FPSSU96] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

[GRG98] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - A framework for fast decision tree construction of large datasets. VLDB 1996.

[Han97] D.J. Hand. *Construction and Assessment of Classification Rules*. John Wiley & Sons, Chichester, England, 1997.

[LLS97] Tjen-Sien Lim, Wei-Yin Loh, and Yu-Shan Shih. An empirical comparison of decision trees and other classification methods. Technical Report 979, Department of Statistics, University of Wisconsin, Madison, June 1997.

[LS97] Wei-Yin Loh and Yu-Shan Shih. Split selection methods for classification trees. *Statistica Sinica*, 7(4), October 1997.

[Man94] O. L. Mangasarian. *Nonlinear Programming*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 1994.

[MAR96] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.

[MFM+98] Y. Morimoto, T. Fukuda, H. Matsuzawa, T. Tokuyama, and K. Yoda. Algorithms for mining association rules for binary segmentations of huge categorical databases. VLDB 1998.

[MS98] N. Megiddo and R. Srikant. Discovering predictive association rules. KDD 1998.

[MST94] D. Michie, D. J. Spiegelhalter, and C. C. Taylor. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, 1994.

[Mur95] S. K. Murthy. *On growing better decision trees from data*. PhD thesis, Department of Computer Science, Johns Hopkins University, Baltimore, Maryland, 1995.

[Olk93] F. Olken. *Random Sampling from Databases*. PhD thesis, University of California at Berkeley, 1993.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[RS98] R. Rastogi and K. Shim. Public: A decision tree classifier that integrates building and pruning. VLDB 1998.

[SAM96] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. VLDB 1996.

[UBC97] P. E. Utgoff, N. C. Berkman, and J. A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29:5–44, 1997.

[Utg89] P.E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4:161–186, 1989.