Scalable Classifiers with Dynamic Pruning

S K Gupta, D V L N Somayajulu, Jitender K Arora, Vasudha B
Department of Computer Science and Engineering
Indian Institute of Technology, NEW DELHI, INDIA - 110016

Abstract

This paper presents an algorithm to solve the problem of classification for data mining applications. This is a decision tree classifier which uses modified gini index as the partitioning criteria. Presorting technique is used to overcome the problem of sorting at each node of the tree. This technique is integrated with a breadth first tree growth strategy which enables us to calculate the best partition for each of the leaf node in a single scan of database. We have implemented this algorithm using depth first tree growth strategy also. The algorithm uses Dynamic Pruning approach which reduces the number of scans of the database and does away with a separate tree pruning phase. The proof of correctness, analysis and performance study are also presented.

1. Introduction

Millions of databases are being used in business, government administration, scientific and engineering data management and in many other applications. The number and volume of such databases keeps growing rapidly because of more and more powerful and affordable database systems. This ever-increasing quantity of data in every computing environment presents both an opportunity to extract useful information, and a challenge to process the massive volume of data effectively. The extraction of this useful information from an enormous data comes under the field of data mining.

Data Mining means a process of nontrivial extraction of implicit, previously unknown and potentially useful information from the data [7]. The rapid growth of data in the world's databases is one reason for the recent interest in data mining. To perform data mining on these huge databases, the data mining algorithms must be scalable and efficient. The running time of the algorithms must be predictable and acceptable in very large databases. Sampling and focusing are some of the solutions to over come this problem [5].

There are several different data mining problems, based on different kinds of knowledge that we can mine from databases [5]. These problems include mining of association rules, classification rules, clustering, similarity search, mining of path traversal patterns etc.

This paper describes design of a Decision Tree Classifier which uses dynamic pruning and is scalable to a large training set. To evaluate splits, gini index is used with some changes. The algorithm uses a pre-sorting and is implemented using the breadth first tree growth [3] and the depth first tree growth strategies.

2. Classification

Let $C$ be a set of $m$ group labels $\{a, c_2, \ldots, c_m\}$, $A$ a set of $n$ attributes $\{A_1, A_2, \ldots, A_n\}$ and let $\text{dom}(A_i)$ refer to the set of possible values for attribute $A_i$. We are given a set of records $T$ in which each object is $(n + l)$-tuple of the form $(i_1, i_2, \ldots, i_n, c^*)$, where each $i_i \in \text{dom}(A_i)$ and $c^* \in C$. We say $T$ is a training set.

We are also given a database $D$ of objects, in which each object is a $n$-tuple of the form $(v_1, v_2, \ldots, v_n)$, where each $v_i \in \text{dom}(A_i)$. In other words the objects in $T$ have the same attributes as the objects in $D$ and additionally have a group label associated with them.

First the classification function $\gamma$ is determined, using the information given in $T$, where $\gamma$ is

$$\gamma: A_1 \times A_2 \times \ldots \times A_n \rightarrow C$$

The problem is to find the class labels of the objects in $D$ using $\gamma$. Classification has been successfully used in various applications [9, 2, 5].

Various classification algorithms have been designed to tackle the problem by researchers in machine learning and statistics. Since the recent surge of data mining research in the database community, the classification problem is re-examined in the context of large databases. Unlike the researchers in other fields, database researchers pay more attention to the issues related to the volume of data and are concerned with the efficiency of data retrieval mechanisms [6]. Two
major approaches which attracted the attention of researchers are neural networks and decision trees.

Decision tree classifiers are relatively fast compared to other classification methods. They can be translated into simple and easy to understand classification rules, even by endusers. This makes the conversion of rules into SQL queries easy. ID-3 [8], C4.5 [7], IC [2] and SLIQ [3] belong to the class of decision tree classifiers.

3 Scalable Classifier with Dynamic Pruning - SCDP

Although classification has been studied extensively in the past, most of the algorithms designed earlier are suitable for small training sets. In this paper we are presenting a classifier which is scalable to large training sets. The classifier uses a presorting technique which reduces the number of passes over the training set. Dynamic pruning also helps to reduce the number of scans of the training set. Here two algorithms are presented. In the first algorithm the above techniques are integrated with breadth first tree growth strategy [3]. In the second algorithm these techniques are integrated with depth first tree growth strategy.

3.1 Splitting Index

A splitting index is used to evaluate the "goodness" of the alternative splits for an attribute. Several splitting indices have been proposed in the past. The gini index was originally introduced in [10]. It is defined as follows:

\[ gini(T) = 1 - \sum_{j=1}^{n} p_j^2 \]  

where \( p_j \) is relative frequency of class \( j \) in \( T \). The gini index is actually a measure of the "badness". We modify the gini index to newindex, which is a measure of "goodness".

\[ newindex(T) = \sum_{j=1}^{n} p_j^2 \]  

In SCDP, \( mgindex \) is used as a measure of "goodness" for a split. It can be derived from newindex as follows:

Let \( T \) be the training set. Let split \( s \) divide \( T \) into \( T_1 \) and \( T_2 \). Let \( n_1 \) and \( n_2 \) be the number of records in \( T_1 \) and \( T_2 \) respectively. Then

\[ mgindex(T, s) = n_1 \times newindex(T_1) + n_2 \times newindex(T_2) \]  

Since \( mgindex \) is measure of goodness we choose the split with highest value of \( mgindex \) from all possible splits for partition of the training set.

3.2 Data Structures

Following are the data structures used:

**Attribute list:** A 2-D array is created for each attribute of the training data. An entry in the attribute list has two fields; one contains an attribute value and other an index into the class list.

**Class list:** A class list is created for the class labels attached to the records of the training set. An entry in the class list also has two fields; one contains a class label, the other a reference to a leaf node of the decision tree. The \( i \)th entry of the class list corresponds to the \( i \)th record in the training data. Thus, the class list can at any time identify the partition to which a record belongs.

**Nodes of the tree:** Each leaf node of the decision tree represents a partition of the training data. The partition being defined by the conjunction of the predicates on the path from the node to the root.

**Frequency histograms:** To compute the \( mgindex \) for an attribute at a leaf node, we need the frequency distribution of class values in the data partition corresponding to that node. For each numeric attribute, we have two class histograms: the upper histogram and lower histogram. The upper histogram stores frequency distribution of class values in the data partition \( A > v \) of the leaf node and similarly the lower histogram stores frequency distribution of class values in the data partition \( A < v \) of the leaf node. These histograms are arrays where \( i \)th element stores the frequency of \( i \)th class. Initially upper histogram stores the frequency of each class in the corresponding leaf node and lower histogram stores zero for each class. For categorical attributes each leaf node has a histogram, which is a two dimensional array histo. The /usE0[i][j] stores the frequency of class \( j \) for the records in which value of the attribute is mapped to \( i \).

**Best-split array:** The best-split array for each leaf node contains the best value of the \( mgindex \) for each attribute.

**Mark array:** The mark array is maintained for each attribute to determine whether the record in the attribute list belongs to the right or left partition after the split.

**Pmark array:** For each leaf node this array indicates whether the node is permanent or not.
4 Breadth First Tree Growing

In this strategy splits for all the leaves of the current tree are simultaneously evaluated in one pass over the data. The general algorithm is given below. For detailed algorithms refer to [4].

BEGIN
1 Initialize;
   1.1 Presort and Create attribute lists;
   1.2 Create class list;
   1.3 Initialize data structures for
      the root node;
2 While (non permanent leaves present)
   2.1 Evaluate split;
   2.2 Update class list;
   2.3 Remove permanent leaf nodes;
Endwhile;
END;

Initializing
For each attribute, the data is sorted and attribute list and class list is created. The frequency histograms are initialized.

Evaluating Splits
Attribute lists are processed one at a time. For each value \( v \) in the attribute list of the current attribute \( A \), we find the corresponding entry in the class list using index. This entry yields the corresponding class and the leaf node. If \( A \) is a numeric attribute, we increment the frequency of this class in the lower histogram and decrement in upper histogram of the corresponding leaf. We calculate the mgindex at value \( v \) using lower and upper histograms. For a categorical attribute, we increment the frequency of the corresponding class for corresponding value of the histogram of the concerned leaf node. If \( A \) is a categorical attribute we wait till the attribute list has been completely scanned and then find the best split. In one traversal of an attribute list, the best split using this attribute is completely known for all the leaf nodes. Thus one traversal of all the attribute lists yields the best over all split of all the leaf nodes. The information about best split for all the leaf nodes is saved.

Updating Class List
Inputs to this procedure are lower and upper histograms and best splits found for each of the leaf nodes. We traverse attribute list of the attribute \( A \), involved in the best split. For each value \( v \) of attribute list we find the corresponding index \( i \) to the class list and hence the class and leaf node \( 1 \). If the best split of this leaf node is due to attribute \( A \) (suppose \( A < v_i \) for numeric attribute and \( A \notin \{v_1, v_2, ..., v_m\} \) for categorical attribute), we set the value of mark\([i]\) to '1'. The setting of mark\([i]\) to '1' (or 'r') means this record belong to left (or right) child of the leaf \( 1 \). Thus the information about partitions is stored in the mark array.

Dynamic pruning is done by a procedure which returns 'p' if the data set is permanent else it returns 'n'. It takes the frequency distribution of each class in that data set as input.

At the end the class labels have to be updated in the class list. The class list is traversed with the help of an attribute list. The algorithm uses the "mark array" and "pmark array" to change the labels of leaf nodes to permanent and new leaf nodes.

Remove The Permanent Nodes
While growing the tree, the above steps of splitting nodes and updating labels are repeated until each leaf node becomes a permanent node and no further splits are required. Some of the nodes may become permanent earlier than others and it may be better to condense the attribute list by discarding the entries corresponding to records belonging to these permanent nodes. Note that we do not condense the class list because any change in the class list must be reflected in the indices of all the entries of attribute lists, which costs a lot. Because of this reason the number of entries in the attribute lists and the class list may be different. Therefore we use an attribute list to traverse the class list.

5 Depth First Tree Growth

In breadth first tree growing strategy we repeatedly split the nodes and update class labels, until each of the leaf nodes become permanent and no further splits are required. The problem with this strategy, is that we can remove the records belonging to the permanent nodes in the next scan only, which costs more. The depth first tree growth strategy overcomes this problem. The broad algorithm is given below:

Partition(Data Set S)
BEGIN
1 IF all points in S are in the same class THEN RETURN;
2 bestsplit = Eval_Split();
3 make_partition(mark);
4 partition(S1);
5 partition(S2);
END;

Evaluating Splits
In depth first growth we evaluate best splits for one leaf node at a time. We have two histograms upper and lower as described in the earlier section. The attribute lists are processed one at a time. For each value \( v \) in the attribute list of current attribute
A, we find the corresponding entry in the class list using index, which yields the corresponding class label. The leaf labels will be same for all the the records because we are processing only one leaf node at a time.

If A is a numeric attribute we increment the frequency of this class in lower histogram and decrement in the upper histogram. We calculate the mgindex at value v using lower and upper histograms. For categorical attribute we increment the frequency of the corresponding class for corresponding attribute value. Thus in one traversal of all of the attribute lists, best overall split is known. The algorithm is similar to the breadth first growth except that we evaluate split only for one leaf node at a time.

The procedure of update processing for depth first strategy is again similar, but in this case we do not traverse all the attribute lists. In fact we traverse only the list of attribute involved in the best split.

**Making Partition** This procedure partitions the training set T into \( T \setminus T_s^i \). The input to this procedure is mark array. We traverse all the attribute lists. For each entry j, of the attribute list A, we find the corresponding entry i in the class list. Now if the mark[i] is '1' then we include entry j in the attribute list \( A_j \) of new training set \( T_s \) else we include it in the attribute list \( A_j \), of new training set \( T_s^i \). Note that the attribute lists in training sets \( T \setminus \) and \( T_s^i \) are already sorted.

### 6 Dynamic Pruning

Separate tree pruning phase is acceptable for generation of a decision tree classifier, if the training set is small. But in typical data mining applications, the size of the training set is large, which demands reduced number of scans of training set for the sake of speed. The dynamic pruning approach used in SCDP, is an attempt in this direction. It is described as follows:

Let the total number of records in the data set T be \( n \). Let \( \text{MINCONF}(<1) \) be a user defined threshold. We say

\[
\text{if } ((\text{maxfreq}/n) > \text{MINCONF}) \text{ then } T \text{ is permanent}
\]

If the training set is permanent then we do not split it further and this becomes the permanent leaf node of the tree.

**Example**: In figure 1(a) there is one attribute Age and two classes A and B. The record marked with * is a statistical fluctuation. If we use separate tree building and tree pruning approach the resultant tree is shown in fig 1(c). After tree pruning the tree will be reduced to tree as shown. If dynamic pruning approach is used with \( \text{MINCONF} = 0.9 \), then we get the tree shown in figure 1(b).

### 7 Analysis

#### Correctness

We show that "the tree generated by the algorithm represents a function." First we elaborate on how the classification rules are generated from the tree and then we will prove that these classification rules represent a function. The methodology of generating the tree, makes the following points clear:

1. Each internal node i of decision tree is labeled with a label \( Li \) of the form \( (A_i \ (j) V_i) \) where \( A_i \) is an attribute , \( \langle f \rangle \) is \( \langle \text{or} \ \& \text{for categorical attribute} \rangle \) and \( V_i \) is a value from the \( \text{domain}(A_i) \) (or \( V_i \) is a subset of \( \text{domain}(A_i) \) if \( A_i \) is categorical attribute)

2. Each leaf node is labeled with a class label \( \emptyset \).

Let the complement of operator \( \langle f \rangle \) be \( \langle \neg f \rangle \). Let the labels of the nodes on the path from root to a leaf node i be \( i_1 \langle f \rangle i_2 \langle f \rangle ... \langle f \rangle i_m \) where \( i_m \) is i. Then the classification rule for leaf node i is as follows

\[ A_j \cap A_j = \cap \text{Ci} \]

where \( \text{Ai} \cap \neg \text{Ai} = \text{Li} \) if \( ij+i \) is left child of \( ij \) and \( \text{Aj} \cap \neg \text{Aj} = \text{li} \) if \( ij+i \) is right child of \( ij \)

The set of all the rules represent a relation \( R \) which maps a n-tuple to a class label. To prove that it is a function we must show that it is many to one relation and each n-tuple in the domain of relation is mapped to some class label in \( C \). It is easy to show that every tuple in the domain of \( R \) will be mapped to some value in the \( C \). We show here that it is a many to one relation. We will prove this by contradiction.

Let \( R \) be not a many to one relation, i.e. there exists a tuple \( t = (v_i \ v_2 \ v_m) \) in the \( \text{dom}(R) \) which is related to two class labels of \( C \). So there will be at least two classification rules whose left hand sides will give 'true' for tuple t. Let these two rules are corresponding to leaf node i and j, and these rules are:

\[ \lambda_i \text{ and A}_i \text{ and } ... \text{ and A}_m \text{ and } \lambda_j \text{ and A}_j \text{ and } ... \text{ and A}_m \]

Because both the leaf nodes are different, so these paths will also be different. Let starting from root the first node at which paths change be 1. So

\[ \text{if } \lambda_i = (A_i \langle f \rangle V_i) \text{ then } \lambda_i = (A_i \langle f \rangle V_i) \]

i.e. only one of them will be 'true' for the tuple t and other will be 'false'. This means left hand side of one classification rule is false giving a contradiction.

#### Computational Complexity

The problem of evaluation of best splits for categorical attributes has been shown to be NP complete [2]. The following assumptions have been made for evaluating the complexities.
1. Our first assumption is the number of possible values for all the categorical attributes are less than 10. The total possible splits are less than $2^{10}$, which can be taken as a constant. Without loss of generality we can say that all the attributes are numerical attributes.

2. The best case for the algorithm is that the given training data is permanent, and analysis of this is very trivial. For best case analysis we assume that a node becomes permanent only when it has only one record.

3. Let $n, a, c$ be the number of records, number of attributes and number of classes respectively.

   The worst case occurs when the split at each node divides the data such that one of the children has only 1 record. In each step we are breaking the problem of size $n$ into problems of size $n - 1$ and 1. In this step we have to traverse $a$ attributes. In traversal of each attribute we have to evaluate $n - 1$ splits and each evaluation of a split takes $O(c)$ times. The recurrence relation is:

   \[ T(1) = ac \]
   \[ T(n) = T(n - 1) + (acn) \]
   \[ \text{i.e. } T(n) = O(acn^2) \]

   giving the worst case complexity.

   The best case occurs when split at each node divides the data into two equal parts. In each step we break the problem of size $n$ into two problems of size $n/2$ each. The recurrence is:

   \[ T(1) = ac \]
   \[ T(n) = 2T(n/2) + acn \]
   \[ \text{i.e. } T(n) = O(ac\ln n) \]

   giving the best case complexity.

8 Performance Study

We present the performance of the algorithms for different cases based on number of records, number of classes, number of attributes, minimum confidence and memory available. We have implemented three algorithms, first the algorithm without remove permanent technique, second with remove permanent technique and third the depth first strategy. All of the three algorithms have been implemented in C++ under Unix environment. BFG, BFGRP and DFG stands for Breadth First Search, Breadth First tree growth with Remove Permanent and Depth First tree Growth respectively. The data used for testing and performance analysis is generated by a program developed at IBM [1].

**Number Of Records:** All training sets for this study have 10 attributes, 2 classes. The MINCONF is set at 0.9. The graphs between number of records and execution time is shown in figure 2(a). We see that as the number of records increases, the execution time also increases. There are some exceptions due to the random nature of data. We can see that BFG is slower than BFGRP and DFG. Times taken by BFGRP and DFG are comparable, but DFG is slightly better.

**Number Of Attributes:** The training set used for this study has 1000 records, 2 classes and the MINCONF is set at 0.9. This graph is shown in figure 2(b). We can see that as the number of attributes increases, the execution time also increases except in some cases, which is due to random nature of data. But we can see that execution times of BFGRP and DFG are comparable and better than BFG.

**Number Of Classes:** To study performance with respect to number of classes, we take a training set, which has 20,000 records and 10 attributes. The training set has only 2 classes. We do not increase the actual number of classes but the program treats as if there are different number of classes. As the training set is same, and the program's execution time is proportional to the number of classes. We can see nearly straight lines for all three algorithms figure 2(c). Again here the execution times of BFGRP and DFG are comparable and better than BFG.

**Minimum Confidence:** The training set has 20,000 records, 10 attributes and 2 classes. The graph is shown in figure 2(d). As there are only 2 classes, the execution time does not vary for $MINCONF < 0.6$, but after that it increases rapidly. We can also see from the
Available Memory: To study the performance of BFG and BFGRP with respect to available memory, the training set has 10000 records, 10 attributes, 2 classes and MINCONF = 0.9. We are simulating the size of memory. According to available memory we perform some disk input/output operations. We can see (figure 2(e)) that as we increase the memory size the execution time decreases rapidly. This is due to disk input output operations which we have to perform due to small size of memory. This time becomes constant when the training set starts to fit in the memory. There are large differences between execution times of both the algorithms.

9 Conclusions and Future Extensions

The presorting technique used in SCDP makes it a better algorithm than those decision tree classifiers, which require sorting at each of the internal node. This algorithm does require shifting and removing the permanent records. This appears to be an expensive proposition, but it actually reduces the size of the training set. In a large data base, where the training set size is also large, this reduction may help in saving of disk input/output operation. The Breadth first strategy is found to be slightly inefficient compared to the Depth first strategy. This is because, in breadth first strategy we remove the records corresponding to permanent nodes in the next scan of training set while in depth first strategy we do not traverse such records. Dynamic pruning reduces the scans of the training set.

The algorithm does not handle the categorical attributes efficiently. Although this problem has been proved to be NP-complete [2], one can try to solve this problem by use of some heuristic to find the near optimal result in polynomial time.

The algorithm calculates splitting index for every splitting point. One can explore a sampling technique, where the splitting index is calculated for the sampled splitting points. This will speed up the algorithm. The effect of the sampling on the accuracy of classification can also be studied.

References

[7] P. S. Y. Ming-Syan Chen, Jiawei Han. Data mining: An overview from database perspective. IEEE TKDE, 8(6), Dec 1996.