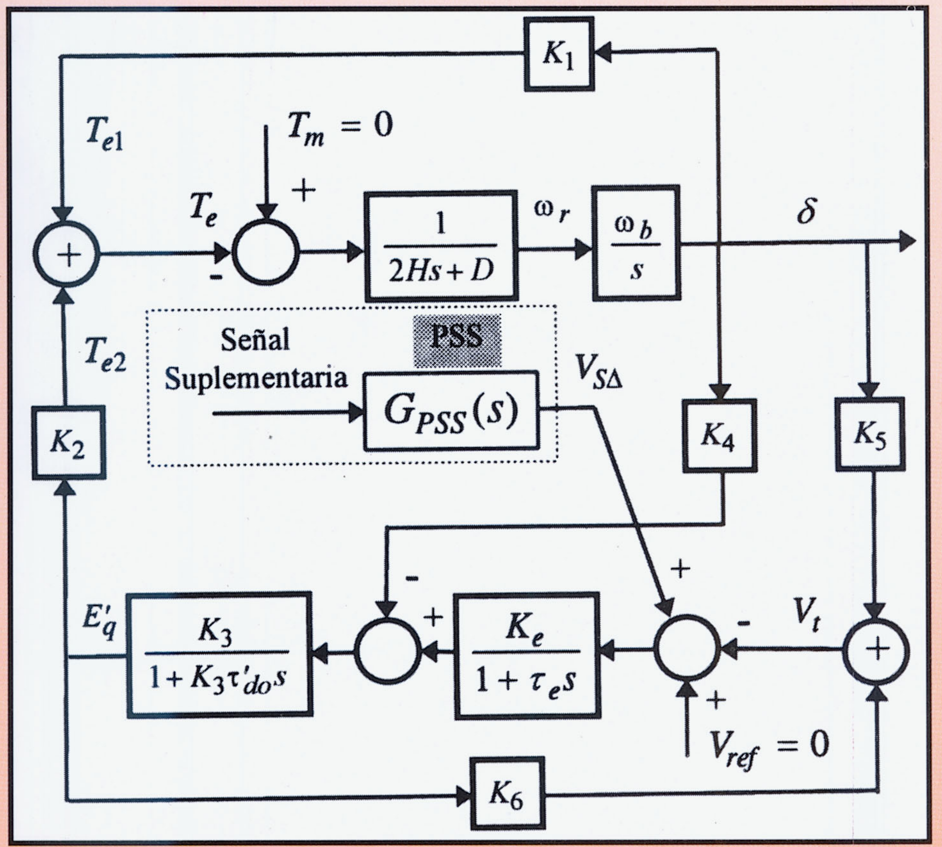
ISSN 1409-2441

Ingeniería

Revista de la Universidad de Costa Rica

Enero/Junio 1997 VOLUMEN 7 N° 1



ISSN 1409-2441

INGENIERIA

Revista Semestral de la Universidad de Costa Rica
Volumen 7, Enero/Junio 1997 Número 1

CORRESPONDENCIA Y SUSCRIPCIONES

Editorial de la Universidad de Costa Rica
Apartado Postal 75
2060 Ciudad Universitaria Rodrigo Facio
San José, Costa Rica

CANJES

Universidad de Costa Rica
Sistema de Bibliotecas, Documentación e Información
Unidad de Selección y Aquisiciones-CANJE
Ciudad Universitaria Rodrigo Facio
San José, Costa Rica

Suscripción anual:
Costa Rica: ¢ 1 000,00
Otros países: US $ 25,00

Número suelto:
Costa Rica: ¢ 750,00
Otros países: $ 15,00

Revisión Filológica: *Lorena Rodríguez*

Diseño Gráfico, Diagramación y Control de Calidad:
Sergio Aguilar Mora

Impreso en la Oficina de Publicaciones de la Universidad de Costa Rica

Revista
620.005
I-46i Ingeniería / Universidad de Costa Rica. ——
Vol. I, no. 1 (ene./jun. 1991)— . —— San José, C. R. : Editorial de la Universidad de Costa Rica, 1991— (Oficina de Publicaciones de la Universidad de Costa Rica)
v. : il

ISSN 1409-2441

Semestral.

1. Ingeniería - Publicaciones periódicas.

CCC/BUCR—250

Ingeniería Vol. 7 Nº 1 - 1997: 21-35; San José, Costa Rica

EXTENSIONS TO DECISION TREE ALGORITHMS FOR CLASSIFICATION AND DATA MINING IN LARGE AND DISTRIBUTED DATABASES

José Ronald Argüello [1]
Sharma Chakravarthy [2]

Resumen

Este artículo describe algoritmos clásicos y eficientes para construir árboles de decisión para muestras de datos , ya sea directamente o incrementalmente, y como mejorarlos con extensiones algorítmicas para hacerlos escalables y útiles para su aplicación en conjuntos grandes de datos. Así, los hacemos útiles para minería de datos en grandes bases de datos. Primero, mostramos los algoritmos básicos y sus problemas principales. Despúes, presentamos nuevas modificaciones que los hacen adecuados para grandes conjuntos de datos. Segundo, mostramos algoritmos distribuidos para tratar con grandes y distribuidas bases de datos.

Abstract

This paper describes classical and efficient algorithms to build decision trees for samples of data, either directly or incrementally, and how to enhance them with algorithmic extensions that make those scaleable and useful for their application to larger sets of data. Thus we make the algorithms useful for data mining in very large databases. First, we show the basic algorithms and their primary problems. Then, we introduce some new modifications to make them suitable for larger data sets. Second, we show distributed algorithms to deal with large and distributed data bases.

1- INTRODUCTION

The basic algorithm for decision tree induction was introduced by J.R. Quinlan [Michalski:83] [Quinlan:86]. Incremental solutions based on tree restructuring techniques were introduced by Schlimmer and Utgoff [Schlimmer:86], [Utgoff:89]. Those algorithms requires one pass over previously seen data per level in the worst case, as does Van de Velde's incremental algorithm, IDL, based on topologically minimal trees [Van-de-Velde:90]. First, we show those classical algorithms and then we discuss enhancements to make them suitable for larger data sets and distribute ones.

2- THE CENTRALIZED DECISION TREE INDUCTION ALGORITHM

Quinlan's traditional algorithm for decision tree induction [Michalski:83, pp 469] was as follows:

Decision Tree Induction Algorithm:

```
[(s1)]     Select a random subset of the given instances (the window)
[(s2)]     Repeat
[(s2.1)]       Build the decision tree to explain the current window
[(s2.2)]       Find the exceptions of this decision tree for the remaining instances
[(s2.3)]       Form a new window with the current window plus the exceptions to the decision tree generated from it
           until there are no exceptions
```

Step 2.1 is called *Decision Tree Derivation* and step 2.2 is called *Decision Tree Testing*.. Step 2.3 is the major drawback in the above algorithm since it forces the process to pass over

[1] Professor University of Costa Rica
[2] Professor University of Florida

all the training data (the window) again, and therefore the algorithm is not incremental.
The algorithm presumes that none of the instances are stored within the decision tree thus preventing the algorithm for being incremental, and also assumes that no additional information is needed in each node besides the decision data.

The *Decision Tree Derivation* (step 2.1) proceeds in two stages --a selection stage followed by a partition stage:

Derivation Algorithm:

```
[(s2.1.0)] If all instances are of the same class,
           the tree is a leaf with value equal to
           the class, so no further passes are
           required.
[(s2.1.1)] Select the best attribute (the root)
           according to a criterion - usually
           statistic
[(s2.1.2)] Split the set of instances according to
           each value of the root attribute.
[(s2.1.3)] Derive the decision subtree for each
           subset of instances.
```

Steps s2.1.1 and s2.1.2 of this algorithm, the selection and partition steps, respectively, each require one pass over the data set. Selection steps usually count the relative frequency in the data set of every attribute-value with the class value (Class counts) which are then used statistically to compute the best attribute (the root). The partition steps distribute the data across the different branches of the root attribute. Thus, the algorithm in general requires **two passes** over the data **per level** of the decision tree in the worst case.

3- THE INCREMENTAL ALGORITHMS

As mentioned above, the incremental algorithm, originally devised by Schlimmer [Schlimmer:86] and Utgoff [Utgoff:89], avoids passing unnecessarily over previously seen instances. To achieve this, it is necessary to keep all Class counts in every node of the decision tree, and it is also necessary to create a mechanism to access previous cases at all leaves of the decision tree for restructuring the tree during the incremental phase. This mechanism is omitted in most implementations since it is assumed that all instances (data base) will be kept memory resident. All previous algorithms start with an empty tree and gradually modify its structure according to the input instances. For every new instance, there is a potential cost of one pass per level over all seen instances.

This cost is half of the cost of directly deriving a tree for traditional algorithms. Hence the importance of the incremental version.

The algorithm below will derive the tree for a part of the database and then update it incrementally (the updating phase) using one instance at a time.

[Incremental Induction Algorithm:]

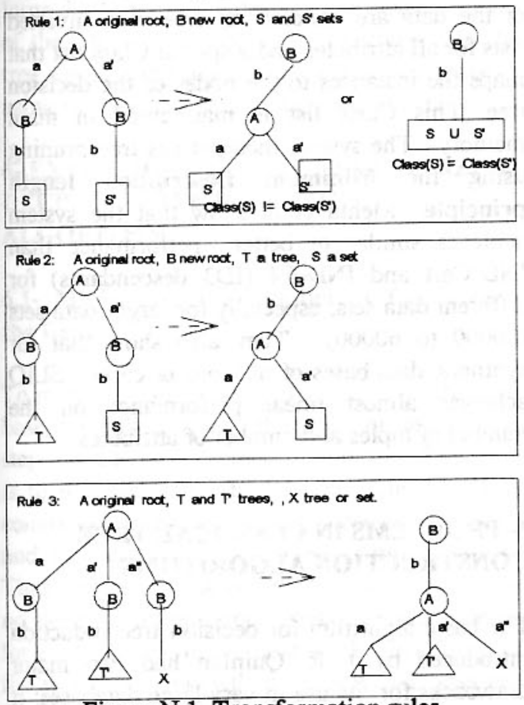
```
[(s0)]   Select a random subset of the data base
         (the window)
[(s1)]   Build the decision tree to explain the
         current window (the Tree) -keep Class
         Counts in every node.
[(s3)]   While there are exceptions; do
[(s3.1)]     Find a exception of the decision tree
             in the remaining instances.
[(s3.2)]     Update the decision tree Class
             counts per node using this
             exception.
[(s3.3)]     Reorganize (Tree); See below.
         done.
```

Incremental algorithms usually start with a random subset of one element. The algorithm above doesn't preclude this possibility.

3.1- Tree Reorganization

Tree reorganization is the key for incremental algorithms including algorithms which are not based on statistics over the input instances [Van-de-Velde:90]. This technique is essential to avoid traversing the whole data base again when dealing with very large databases.
Hopefully, tree reorganization will require just a small part of the database when the tree is restructured.

The reorganization part depends on the relative representation suited for the algorithm.
Utgoff maps every attribute-value pair to a new boolean attribute [Utgoff:95]. Thus, he assumes all trees are binary trees.


Rule 1: A original root, B new root, S and S' sets
A
a
a'
B
B
b
b
S
S'
-->
B
b
A
a
a'
S
S'
Class(S) != Class(S')
or
B
b
S U S'
Class(S) = Class(S')
Rule 2: A original root, B new root, T a tree, S a set
A
a
a'
B
B
b
b
T
S
-->
B
b
A
a
a'
T
S
Rule 3: A original root, T and T' trees, , X tree or set.
A
a
a'
a"
B
B
B
b
b
b
T
T'
X
-->
B
b
A
a
a'
a"
T
T'
X
Figure N 1. Transformation rules

Tree reorganization algorithms restructure the tree when a better attribute is detected (or inherited in the case of a subtree).
The basic idea is to force all subtrees to keep the same root (the best attribute) and then apply a transformation rule to exchange the actual root of the tree with each subtree (See figure 1 and algorithm below). In this way, some subtrees are pruned when all subtree branches lead to the same class value.

The ID5R pull up algorithm reorganizes the decision tree in theway just mentioned [Utgoff:89]. If a tree is just a leaf (a set of instances) , the pull up algorithm assumes the respective attribute as the root of the decision tree starting on that leaf. Then the leaf is **expanded** i.e., the decision tree is built.

```
[ The ID5R pull up algorithm ]

[(s1)]    If the attribute A to be pulled up is
          at the root, then stop.
[(s2)]    Otherwise,
[(s2.1)]    Recursively pull up the attribute A
            to the root of each immediate
            subtree.
[(s2.2)]    Transpose the tree, resulting in a
            new tree with A at the root, and
            the old root attribute at the root of
            each immediate subtree.
```

Note that in step s2.2 the transformation rules of figure 1 must be applied to obtain the transposed tree.

Van de Velde 's algorithm IDL uses the same pull up technique for reorganization as ID5R [Van-de-Velde:90]. IDL differs from others in that it uses a topological criterion, called *topological relevance gain*, based on the tree structure to select the actual root attributes for the subtrees. Van de Velde shows how his algorithm is able to discover concepts like the tree shown in figure 2 ; while traditional algorithms fail to discover this tree.

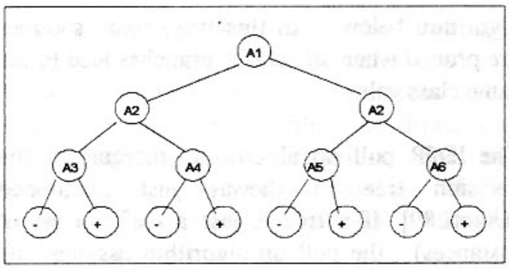


Figure N 2. A tree for the 6-multiplexer

Basically, the topological relevance $TR_m(A,e)$ criterion measures the number of occurrences of a given attribute A for a given example e when this is used to traverse the tree starting from any leaf of the example class all the way up until the node m if it is possible. It depends uniquely on the actual tree structure and the given example. Thus, given nodes m and s in the classification path of an example e, with s the immediate son

$$TRG_m(A,e) = \frac{TR_m(A,e) - TR_s(A,e)}{TR_m(A,e)}$$

of m, the *topological relevance gain* for attribute A is:

When compared to its predecessor ID5R, IDL saves computations costs in terms of class counts, criteria computations, expansions of sets , pruning and transformations; while keeps better or similar accuracy.

More recently, Utgoff has implemented the ITI algorithm which is a direct descendant of ID5R and uses reorganization-like techniques in a similar way [Utgoff:95].

4- OTHER APPROACHES

SLIQ , a fast scalable classifier for Data Mining was designed to solve the classification problem for Knowledge Discovery [Mehta:96]. Conceptually, the SLIQ system uses the same algorithm, where the selection criterion is the gini index - a criterion that splits the range of numerical attributes in two parts. It also uses set splitting for categorical attributes. The gini index for a set S containing n classes is :

$$\boxed{G(S) = 1 - \Sigma p_j^{\,2}}$$

where p_j is the relative frequency of class j, and then the attribute measure is:

$$\boxed{G(A=a) = P(A \leq a) * G(A \leq a) + P(A > a) * G(A > a)}$$

where $A \leq a$ or $A > a$ represents the set of tuples that satisfies the relation.

Thus, SLIQ representation is a binary decision tree. In order to make the system scalable, most of the data are handled off line with inverted lists for all attributes and a special Class list that maps the instances to the nodes of the decision tree. This Class list is maintained in main memory. The system incorporates tree pruning using the **Minimum Description length principle** Mehta et al show that the system achieves similar or better performance than IND-Cart and IND-C4 (ID3 descendants) for different data sets; especially for larger data sets (20000 to 60000). They also show that for synthetic data bases of millions of cases, SLIQ achieves almost linear performance on the number of tuples and number of attributes.

5- PROBLEMS IN CLASSICAL TREE CONSTRUCTION ALGORITHMS

The basic algorithm for decision tree induction introduced by J. R. Quinlan had two major drawbacks for its use in very large databases: it was not incremental and it required, in the worst case, two passes over the entire data *per level* to build the decision tree [Quinlan:86].

The incremental solution based on tree restructuring techniques [Schlimmer:86] [Utgoff:89] requires one pass over previously seen data *per level* in the worst case, as does

Van de Velde's incremental algorithm IDL based on topologically minimal trees [Van-de-Velde:90]. This makes the utility of the incremental version more attractive for large databases. However, the incremental version requires keeping the data ``inside" the decision tree structure [Utgoff:95] in main memory and hence it is likely to have a high cumulative cost [Mehta:96], which partially precludes its use for large databases. This section describes a *one-pass per level* worst case algorithm to build the tree for a sample of data, which makes it equal to or even better than building the tree incrementally. In either case, the expected number of nodes of a decision tree in very large databases requires a mechanism to store part of the tree in external memory. Since the size of large databases precludes keeping several copies of the data, data can be incorporated into the tree leaves using indices to the main database or using the tree as a way to fragment the database.

6- EXTENSIONS TO THE CENTRALIZED DECISION TREE INDUCTION ALGORITHM

6.1- Minimizing the Number of Passes over the Data

To minimize the number of passes over the data base, the split step and the selection of the next step need to be combined in one pass. The trick is to use each case (tuple) to update the Class counts of the corresponding subtree (or subset) and to create the data subset simultaneously. Thus, in the next selection step, there will be no need for an additional pass over the subsets for every subtree in the next level. Then, even in the worst case, we will need only one pass per level over the data base.

The first step of the derivation must proceed like this:

```
[Derivation Revisited (Initial step) ]

[(s2.1.0)] **If** all instances are of the same class,
           the tree is a leaf with value equal to
           the class, so no further passes are
           required.
[(s2.1.1)] **Select** the best attribute (the root)
           according to a criterion - usually
           statistic
[(s2.1.2)] **Split** the set of instances according to
           each value of the root attribute.
           Update Class Counts for every subtree
           with each instance
[(s2.1.3)] **Derive** the decision subtree for each
           subset of instances.
```

Then, for each subtree:

```
[ Derivation Revisited ]
[(s2.1.0)] **If** all instances are of the same class,
           the tree is a leaf with value equal to
           the class, so no further passes are
           required.
[(s2.1.1)] **Get** the best attribute (the root)
           according to a criterion - usually
           statistic
[(s2.1.2)] **Split** the set of instances according to
           each value of the root attribute.
           Update Class Counts for every subtree
           with each instance
[(s2.1.3)] **Derive** the decision subtree for each
           subset of instances.
```

Note that the initial step requires two passes to check the data. After that, the remaining steps require just one pass per level. The selection step does not require a pass over the data since all Class Counts were computed previously.

The merging of these two steps is not without cost. Additional memory is required to keep all frequencies (Class counts) for every subtree. If we keep all those frequencies in memory, then it is clear that the decision tree can be built for every subtree, without additional disk accesses. Note that only the class counts for the last level are needed and that the number of counts maintained in main memory are fewer whenever the level (of the tree) is higher. However, there can be thousands of leaves in a decision tree for a large data base. Eventually, a mechanism to keep the class counts outside of main memory is needed. But even if this is done for every subtree, additional disk accesses will be incurred for constructing each subtree. The number of additional disk accesses for reading class counts will in general be lower than the number of disk accesses required to read the whole subset. A threshold mechanism to avoid incurring these overhead costs for small data sets can easily be implemented.

6.2- Improving the Halting Criteria

The Tree Derivation Algorithm halts when all instances in the data set are from the same class (step 2.1.0). It is impractical to expect this since data can be inconsistent or incomplete in the sense that there are not enough attributes to correctly classify the data.
Thus, a threshold criterion must be introduced to stop the process when the *set measure* is beyond a certain point.
The set measure corresponds to the same statistic used to evaluate and select attributes in step 2.1.1.

Quinlan's algorithm assumes that if all data are not from the same class, the attribute selection step will improve the classification. The following case shows this is not necessarily true. Suppose we have two classes with a distribution of 90% for positives and 10 % for negatives. Assume that every attribute splits the set in two halves, each one with 45% positives and 5% negatives. The best selected attribute will be either of them; but the average measure will be the same since the relative distribution of classes in each leaf is the same as it is in the original data set. The information expected criterion will give us 0.47 entropy (0.53 certainty) in both cases. As the result is equal to the set measure, no improvement has been made.

Even though, the previous example is an extreme case, usually absent in practice, the algorithm must check for this condition. In general, the algorithm must check if the **average certainty** is below or equal to the **set certainty**.

For the Determination measure, the conjecture is not true. For example, with 90% positive cases and 10% negative cases, (0.88 Determination), the partition in one set of 80% positive and 0% negative, and another of 10% positive and 10% negative, does not lead to a better average determination (0.8 (1) + 0.2 (0) = 0.80).
Note that the entropy (certainty) changes from 0.47 (0.53) to 0.20 (0.80).

This property of the Determination measure will allow us to prune the decision tree before it fits the data unnecessarily since there is no improvement in the measure. On the contrary, the entropy will continue choosing attributes (even if they are irrelevant to the classification) since entropy decreases (certainty increases) with every partition if the previous conjecture is true.

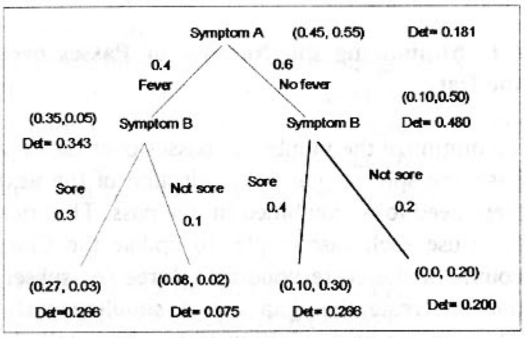


Figure N 3. Determination measures

As an example, consider the tree in figure 3. The same tree with entropy computed measures is shown in figure 4.

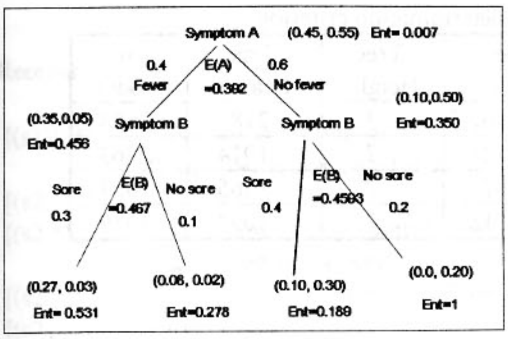


Figure N 4. Entropy measures

Note that the certainty always increases when entropy is used. This tree doesn't need to be built completely when determination is used. The derived tree will be the tree depicted in figure 5. Note that both subtrees starting with root Symptom B were not needed since E(B) was always lower than the respective set determination (det). This coincides with the fact that the determination chooses the most general rule.

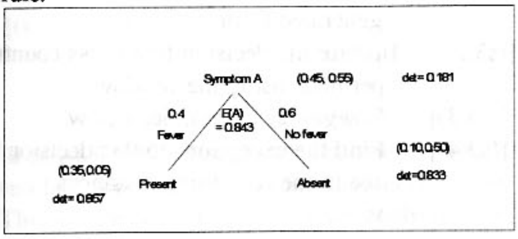


Figure N° 5. Pruned decision tree with determination

6.3-Pruning Using Confidence and Support

Related to the previous section but applicable in a different way is the mechanism to prune the tree. The most general method is called the Minimum Description Length principle introduced by Quinlan [Quinlan:89]. It has been succesfully used in most of the actual systems [Utgoff:95], [Mehta:95], [Mehta:96] . However, it improves the accuracy and reduces the size of the decision tree; the MDL principle is based on the future error and the cost of building the subtree pruned. It is not related to implicit rules or to the user viewpoint. In this sense, the pruning is artificial and of little or not interest to the user and the application.

The confidence and support introduced here allow us to incorporate the end user and the meaning of the rules to be extracted as criteria to prune the tree. The user can specify the thresholds for support and confidence. When the subset cardinality in a leaf is below the minimum support or the confidence in the final classification is greater than a maximum confidence factor, then the tree construction process must be stopped. All potential rules will satisfy the requirements. Note that, unlike the MDL principle, we don't care about the final error or the amount of work needed to build the tree. Our goal is to meet the confidence and support thresholds.

Similarly, the attribute selection criterion gave us a good tool for tree pruning if we can predict the final outcome in terms of confidence or support. Entropy can not be used for this, since there is no way to relate the entropy measure to the set confidence. In my opinion, this is the primary reason for the developing of pruning criteria such as the MDL principle.

Confidence and determination are related by:

$$\boxed{\delta = D(p_1, p_2, \ldots, p_n) = \frac{n\ Conf - 1}{(n-1)\ Conf}}$$

and therefore

$$\boxed{Conf = 1\ /\ n - (n-1)\ \delta}$$

An artificial database with two classes, 5952 cases, 20 attributes plus a class attribute was used to generate a decision tree with different determination levels (confidence levels). The results are shown in table 1. It can be observed that savings until one 50% on the size of the tree was achieved by pruning the tree with 85% determination (87% confidence) without sacrificing largely the error rate (no more than 5%).

Table N° 1. Pruning with the determination criterion

Test	Prune value	Tree Error	Tree Size	Tree Height	Tree Leaves	Tree Nodes
1	0.99	0.0005	2316462	7	2187	2688
2	0.95	0.0087	2035728	7	1914	2363
3	0.90	0.022	1677073	7	1565	1937
4	0.85	0.058	1174018	7	1094	1350

7- EXTENSIONS TO THE INCREMENTAL ALGORITHMS

The incremental algorithm, originally devised by Utgoff [Utgoff:89], avoids passing unnecessarily over previously seen instances.

With our one pass algorithm, it is necessary to re-evaluate the incremental version -- since the cost of both approaches is O(n) in general. However, direct tree derivation is a **pessimistic** approach and assumes Nothing about the data. Incremental algorithms are **optimistic** and they assume that the previous decision tree reflects the actual decision tree. Using this information, the practical performance of the incremental algorithms can be improved as compared to the direct (brute-force) approach. I will discuss more thoroughly the re-organization approach used in incremental algorithms in a later section.

In general, the cumulative cost of the **pure incremental** algorithm -one instance at a time- will preclude its use over a direct derivation algorithm over the data base. The algorithm below will derive the tree for a part of the data base and then update it incrementally (the updating phase) using chunks of wrongly-classified instances instead of one instance at a time.

[Partial Incremental Induction Algorithm:]

```
[(s0)]    Select a random subset of the data base
          (the window)
[(s1)]    Build the decision tree to explain the
          current window (the Tree) -keep Class
          counts in every node.
[(s2)]    Find the exceptions of this decision tree
          in the remaining instances.
[(s3)]    While there are exceptions; do
[(s3.1)]      Form a new window with a portion
              of the exceptions to the decision tree
              generated from it.
[(s3.2)]      Update the decision tree Class counts
              per node using the window.
[(s3.3)]      Reorganize (Tree); See below.
[(s3.4)]      Find the exceptions to this decision
              tree in the remaining instances.
          done.
```

7.1- Tree Reorganization Algorithms

As we discussed above, tree reorganization algorithms restructure the tree when a better attribute is detected (or inherited in the case of a subtree). A more detailed algorithm for tree reorganization is given below. Again, I have based the algorithm on the transformation rules in figure 1.

7.1.1-The Reorganization Algorithm

The reorganization procedure involves two parameters: the actual decision tree (Tree) and the new root attribute (NewRoot).

Reorganize(Tree, NewRoot):

```
[(s1)]  If the NewRoot is null then
              NewRoot = better attribute for Tree.
[(s2)]  If Tree is a leaf,
[(s2.1)]      Create a new tree by splitting the
              set according to the NewRoot
[(s2.2)]      Make Tree equal to this new Tree.
[(s2.3)]      return
[(s3)]  otherwise   (If Tree is not a leaf)
[(s3.1)]      If Tree.Root == NewRoot then
                  return
              otherwise
[(s3.2)]      For each Subtree,
[(s3.2.1)]       Reorganize(Subtree, NewRoot);
[(s3.2.2)]       Apply the transformation rule
[(s3.2.3)]       Update class counts for the
                     subtrees. (now starting with
                     previous root Tree.Root).
[(s3.3)]      For each subtree STree,
                 Reorganize(STree)
[(s3.4)]      return
```

In step s3, the best attribute is bubbled up until it reaches the root of the current decision tree. This is repeated for the next level of the decision tree until all subtrees hold the best attributes as roots or until they are just leaves. There is a potential for doing a pass over the data at the leaves for each level and therefore the algorithm requires one pass per level. However, in practice, we expect that one attribute for the root candidate is already a subroot of a subtree and there is no need to reorganize the subtree. Thus, this algorithm will in general be better than the direct approach if the previous decision tree resembles the actual decision tree, which is likely since the tree was based on a representative subset (a percentage) of the actual data. See example in figure 6.

8- DISTRIBUTED TREE INDUCTION ALGORITHMS

8.1- Distributed Subtree Derivation

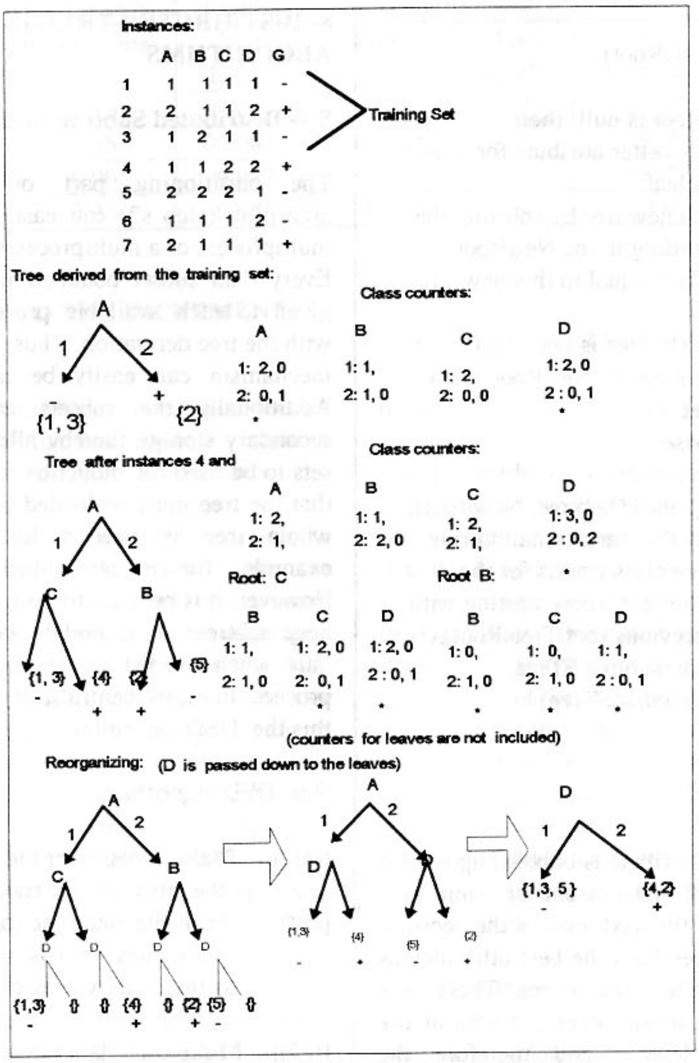
The partitioning part of the Derivation algorithm (step s2) can easily be adapted to a multiprocess or a multiprocessor environment. Every data subset obtained in the partition is given to each available processor to continue with the tree derivation. Thus, the tree induction mechanism can easily be made in parallel. Additionally the subsets can be kept on secondary storage thereby allowing even larger sets to be used for induction with the restriction that the tree must be loaded into memory if the whole tree is needed for processing (for example, for a centralized testing phase). However, it is possible to design a mechanism to keep subtrees in secondary storage and loaded only when needed. The updating phase will proceed like any centralized algorithm. I term this the DSD algorithm.

The DSD algorithm:

```
[(s1)]  Make a pass over the data set to select
          the attribute (the root).
[(s2)]  Split the data base (or create new
          index files) into as many data subsets
          as there are values of the root attribute.

[(s3)]  Make each data subset available for
          other processors (saving one for
          itself).

[(s4)]  While there are subsets,
             apply the DSD to each subset.
[(s5)]  If all data subtrees are ready, then
             make the decision tree, attaching to
             each branch of the root the
             respective decision tree.
[(s6)]  Exit.
```



Instances:

	A	B	C	D	G
1	1	1	1	1	-
2	2	1	1	2	+
3	1	2	1	1	-
4	1	1	2	2	+
5	2	2	2	1	-
6	1	1	1	2	-
7	2	1	1	1	+

Training Set
Tree derived from the training set:
Class counters:
Tree after instances 4 and
Class counters:
Root: C
Root: B:
(counters for leaves are not included)
Reorganizing: (D is passed down to the leaves)

Figure N° 6. Tree Reorganization

In step s3, the relative speed of every available processor can be taken into account or every subset will simply be distributed on a first-come first-served basis. Similarly, in order to fully use the distributed capabilities of the system, a set will be available if its size is greater than a threshold set previously by the user.

The algorithm is useful when several processes or processors can cooperate to help in the decision tree derivation. It is assumed that they at least share a file system. For example, the algorithm can be used when the decision tree does not fit in the memory available for each process or processor.

8.2- Distributed Tree Derivation

An alternative use of distributed processing capability in deriving decision trees is to assume that the training data is already distributed among processors (if not, a first pass can distribute the data equally among available processors). Thus, processors can interchange class counts on every attribute-value pair and then each one will arrive at the same conclusion on the selected attribute as a root. Then, as each data set will be partitioned accordingly, a new interchange of class counts will occur for each possible subtree, until the complete decision tree is derived for each processor.

Communication is reduced to a minimum since data sets are not interchanged, just the attribute-value-class frequencies or Class counts (See Figure 7). This will be called the DTD algorithm.

For this algorithm, each processor has its own data set.

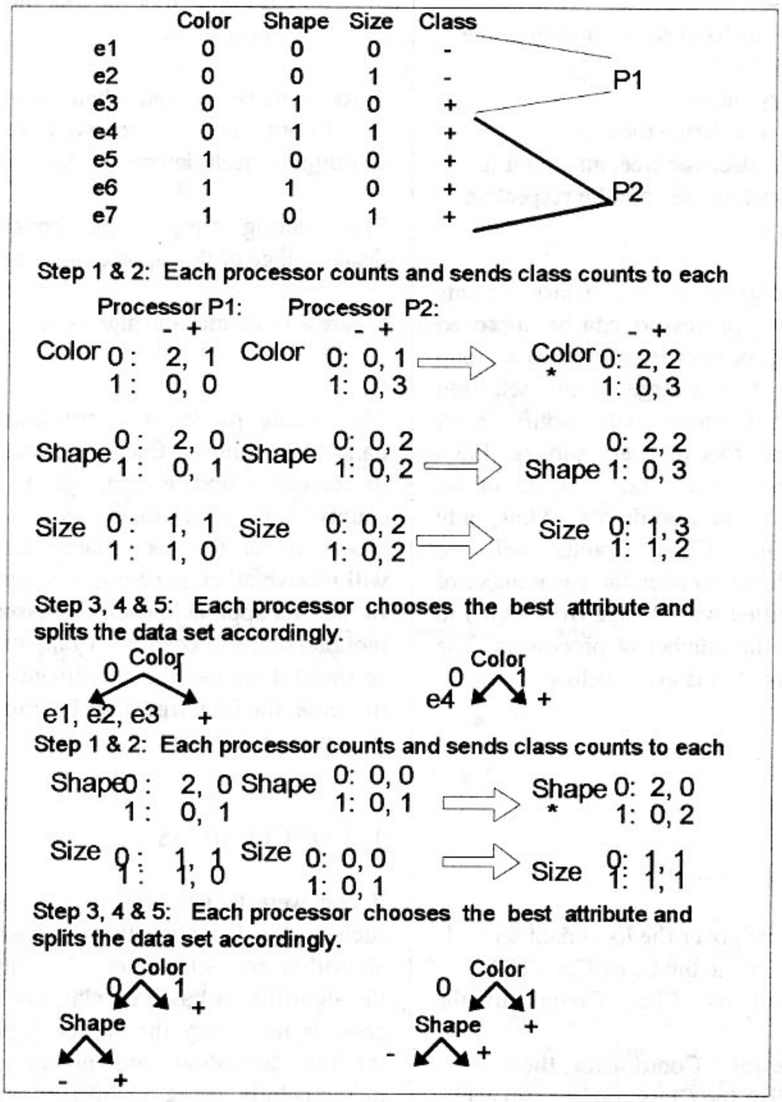


Figure N 7. Distributed Tree Derivation.

The DTD algorithm:

```
[(s1)]  **Make** a pass over the local data set and
        create the Class Counts.
[(s2)]  **Send** the Class Counts to every
        processor.
[(s3)]  **Receive** the Class counts from each
        processor and summarize.
[(s4)]  **Select** the best attribute (the root).
[(s5)]  **If** the tree is a leaf, **then return**
        **otherwise**
            **Split** the local set according to the
            root values
[(s6)]  **For** every subset,
          recursively derive the tree.
[(s7)]  **Make** the decision tree, attaching to
        each branch of the root the respective
        decision tree.
```

In the above algorithm, the Class Counts interchange among processors can be improved significantly if a processor is selected as a group coordinator and is in charge of the selection stage. This Coordinator will notify each processor the next root at every subtree. Each processor will send the Class Counts of its respective subset to the Coordinator. Thus, only one copy of the Class Counts will be transmitted. With the coordinator, the number of messages transmitted will change from $O(n^2)$ to $O(n)$, where n is the number of processors. The revised DTD algorithm is given below:

The Revised DTD algorithm:

```
[(s1)]  Make a pass over the local data set and
              create the Class Counts.
[(s2)]        Send the Class Counts to the
Coordinator.
[(s3)]  **If** Processor = Coordinator, **then**
[(s3.1)]    **Receive** the Class counts from each
            processor and summarize.
[(s3.2)]    **Select** the best attribute (the root).
[(s3.3)]    **Notify** each processor of the root
            selected and next subset to process
[(s4)]  **Wait** until root is defined.
[(s5)]  **If** the tree is a leaf, **then return**
        **otherwise**
            Split the local set according to the
            root values
[(s6)]  **For** every subset,
            recursively derive the tree.
[(s7)]  **Make** the decision tree, attaching to
        each branch of the root the respective
        decision tree.
```

A root will be defined when the message from the Coordinator is received or when the coordinator itself determines the root.

The waiting time could consist a major disadvantage of this approach in step s4.

Figure 8 illustrates the algorithm.

The update phase in a distributed setting is handled as follows. Each processor will receive its respective update data, get its partial class counts and send them over to all other processors or the coordinator. Each processor will receive all class counts and update the tree. In the first approach, each processor will call its reorganization procedure. Computing time will be saved if we use the coordinator approach. In this case, the final tree must be transmitted to all remaining processors.

9- CONCLUSIONS

If one were to use any incremental algorithm, such as the ITI algorithm [Utgoff:95] or the algorithm by Schlimmer [Schlimmer:86] and the algorithm is based on class counts, then it is possible to employ the approach proposed here for the derivation and updating every tree incrementally using chunks of updating data (sending class counts for a single case will be more costly than sending the case data).

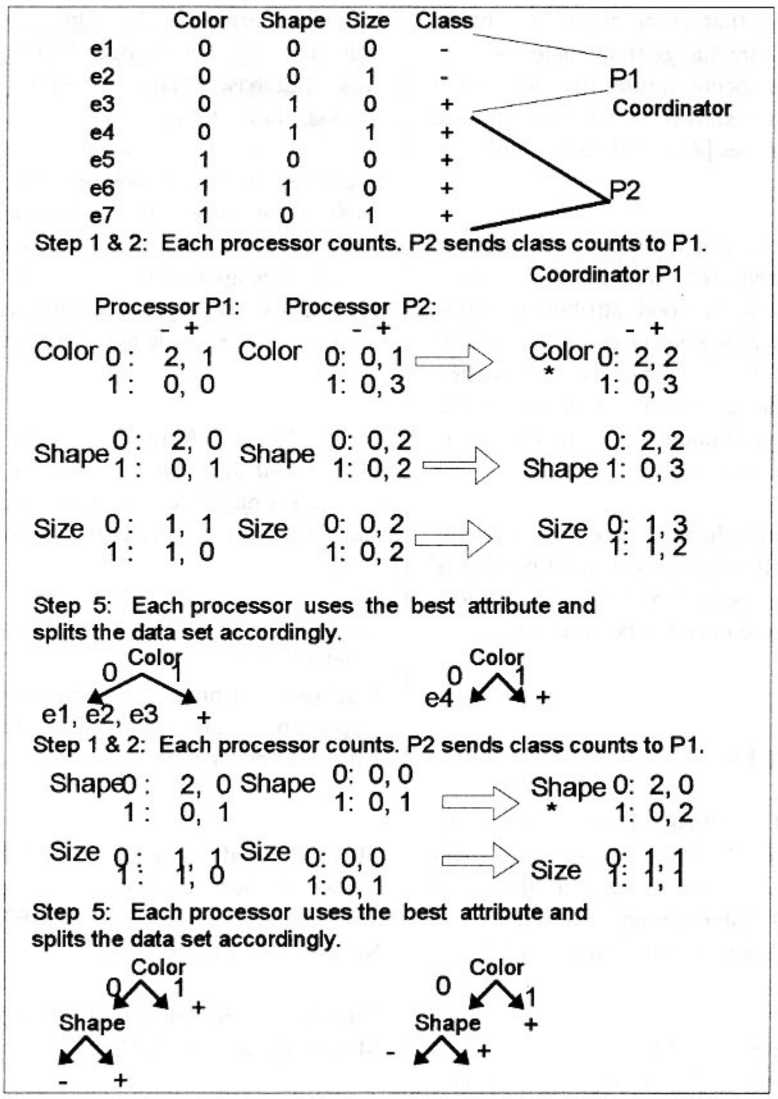


Figure N 8. Revised Distributed Tree Derivation.

To get an estimate of the data to be stored in memory (or secondary storage), consider the following parameters: 25 attributes, 100 values per attribute and 100 classes. Then the array of frequencies will be at most 250,000 entries. The expected universal domain for a database with those parameters will be 10^{25} potential entries. Even small subsets will be big enough to make class count interchange beneficial.

It is clear that if a processor keeps only a few tuples, it will be better to transmit these tuples than to transmit the class counts. However, the receiving processor must compute the frequencies and some time can be saved if one uses the idle processors to do this instead of eventually loading the receiving processor with small computations from different sets. It is worth mentioning that other algorithms based on compute and interchange frequencies or class counts to derive association rules in a distributed environment have shown better performance than other approaches [Agrawal:93,94 , 96].

The applicability of tree induction techniques in large databases requires tree reorganization , incremental procedures, good attribute selection criteria and good grouping of continuos values to minimize the number of branches. Algorithms as those shown are useful when dealing with large amounts of data [Arguello: 96a, 96b] [Grossman et al, 96].

Exhaustive procedures are eventually prohibitive; mostly when only a small percent of the database has been modified and previous rules or classification need to be updated.

10- REFERENCES

Agrawal, R. and Imielinski, T. and Swami, A.. *Mining Association Rules between set of items in largedatabases*, Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, Usa 1993.

Agrawal, R and Shafer, J.C. *Parallel Mining of Association Rules*, Proceedings of EDBT-96, France 1996

Agrawal, R and Srikant, R *Fast algorithms for Mining of Association Rules in large databases*, Proceedings of the 20th International Conference on Very Large Data Bases, Santiago, Chile 1994.

Argüello, J.R and Chakavarthy , S. *Distributed Tree Induction for Knowledge Discovery in very large distributed databases*, First ACM Workshop on Data Mining. Montreal. Canadá. Junio 1996.

Argüello, José Ronald. On Decision Tree Induction for Knowledge Discovery in Very Large Databases Tesis Doctoral. University of Florida. Junio 1996.

Grossman, R. and Bodek, H. and Northcutt, D. *Early Experience with a System for Mining, Estimating, and Optimizing large collections of objects managed using an object wharehouse*, SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, Montreal 1996.

Mehta, M. and Agrawal, R. and Rissanen, J.. *MDL based decision tree pruning*, Proceedings of Int'l Conference on Knowledge Discovery and Data Mining (KDD-95), Montreal, Canada 1995.

Michalski, R. S and Carbonell, J. G and Mitchell, T.M. Machine Learning. An Artificial Intelligence Approach, I, Tioga Publishing Company Palo Alto, California 1983.

Mehta, M and Agrawal , R and Rissanen, J. *SLIQ: A fast scalable classifier for Data Mining*, Proceedings of EDBT 96 France, March 1996, France 1996.

Quinlan, J. R. *Induction of Decision trees*, Machine Learning, 1, 1986.

Quinlan., J. R.. *Inferring decision trees using the minimum description length principle*, Information Computer, 80, 1989.

Schlimmer, J. C. *A case study of incremental concept induction*, Proceedings of AAAI, Philadelphia 1986.

Utgoff, P. E. *Incremental Induction of Decision Trees*, Machine Learning, 4, 1989.

Utgoff, P. E. *Decision Tree Induction based on efficient tree restructuring*, Technical Report. 95-18 Department of Computer Science. University of Massachussetts, 1995.

Van de Velde, W. *Incremental Induction of Topologically Minimal Trees*, Machine Learning: Proceedings of the Seventh International Conference, University of Texas, Austin , Texas 1990. /