An Interval Classifier for Database Mining Applications

Rakesh Agrawal Sakti Ghosh Tomasz Imielinski* Bala Iyer Arun Swami

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

We are given a large population database that contains information about population instances. The population is known to comprise of m groups, but the population instances are not labeled with the group identification. Also given is a population sample (much smaller than the population but representative of it) in which the group labels of the instances are known. We present an interval classifier ($\mathcal{IC}$) which generates a classification function for each group that can be used to efficiently retrieve all instances of the specified group from the population database. To allow $\mathcal{IC}$ to be embedded in interactive loops to answer adhoc queries about attributes with missing values, $\mathcal{IC}$ has been designed to be efficient in the generation of classification functions. Preliminary experimental results indicate that $\mathcal{IC}$ not only has retrieval and classifier generation efficiency advantages, but also compares favorably in the classification accuracy with current tree classifiers, such as ID3, which were primarily designed for minimizing classification errors. We also describe some new applications that arise from encapsulating the classification capability in database systems and discuss extensions to $\mathcal{IC}$ for it to be used in these new application domains.

*Current address: Computer Science Department, Rutgers University, NJ 08903

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

1 Introduction

With the maturing of database technology and the successful use of commercial database products in business data processing, the market place is showing evidence of increasing desire to use database technology in new application domains. One such application domain that is likely to acquire considerable significance in the near future is *database mining* [4] [10] [12] [16] [18] [17] [20]. Several organizations have created ultra large data bases, running into several gigabytes and more. The databases relate to various aspects of their business and are information mines that they would like to exploit to improve the quality of their decision making.

One application of database mining involves the ability to do *classification* in the database systems. Target mailing is a prototypical application for classification, although the same paradigm extends naturally to other applications such as franchise location, credit approval, treatment-appropriateness determination, etc. In a target mailing application, a history of responses to various promotions is maintained. Based on this response history, a classification function is developed for identifying new candidates for future promotions. As another application of classification, consider the store location problem. It is assumed that the success of the store is determined by the neighborhood characteristics, and the company is interested in identifying neighborhoods that should be the prime candidates for further investigation for the location of next store. The company has access to a neighborhood database. It first categorizes its current stores into successful, average, and unsuccessful stores. Based on the neighborhood data for these stores, it then develops a classification function for each category of stores, and uses the function for the successful stores to retrieve candidate neighborhoods.

The problem of inferring classification functions from a set of examples can be formally stated as follows. Let G be a set of m group labels $\{G_1, G_2, \ldots, G_m\}$. Let A

be a set of n attributes (features) $\{A_1, A_2, \ldots, A_n\}$. Let $dom(A_i)$ refer to the set of possible values for attribute A_i. We are given a large database of objects $\mathcal{D}$ in which each object is a n-tuple of the form $< v_1, v_2, \ldots, v_n >$ where $v_i \in dom(A_i)$ and G is not one of A_i. In other words, the group labels of objects in $\mathcal{D}$ are not known. We are also given a set of example objects $\mathcal{E}$ in which each object is a $(n+1)$-tuple of the form $< v_1, v_2, \ldots, v_n, g_k >$ where $v_i \in dom(A_i)$ and $g_k \in G$. In other words, the objects in $\mathcal{E}$ have the same attributes as the objects in $\mathcal{D}$, and additionally have group labels associated with them. The problem is to obtain m classification functions, one for each group G_j, using the information in $\mathcal{E}$, with the classification function f_j for group G_j being $f_j : A_1 \times A_2 \times \ldots A_n \rightarrow G_j$ for $j = 1, \ldots, m$. We also refer to the examples set $\mathcal{E}$ as the *training set* and the database $\mathcal{D}$ as the *test data set*.

This problem has been investigated in the AI and Statistics literature under the topic of *supervised learning* (see, for example, [6] [11] [12] [17]) [1]. We put the following additional requirements, not considered in the classical treatment of the problem, on the classification functions:

1. *Retrieval Efficiency*: The classification function should be able to exploit database indexes to minimize the number of redundant objects retrieved for finding the desired objects belonging to a group. Currently, database indexes can only be used for queries involving predicates of the form $A_i \; \theta \; v$ (point predicates), or $v_1 \; \theta_1 \; A_i \; \theta_2 \; v_2$ (range predicates), or their conjuncts and disjuncts, where θ, θ_1, and θ_2 are appropriate comparison operators.

   Retrieval efficiency has not been of concern in current classifiers, and it differentiates classifiers suitable for database mining applications from the classifiers used in applications such as image processing. In an image processing application (e.g. character recognition), having developed a classification function, the problem usually is to classify a given image into one of the given groups (a character in the character recognition application). It is rare that one uses the classifier to retrieve all images belonging to a group.

2. *Generation Efficiency*: The algorithm for generating the classification functions should be efficient. The emphasis in the current classifiers has been on minimizing the classification error and generation efficiency has not been an important design consideration. This has been the case because usually the classifier is generated once and then is used over and over again. If, however, classifiers were to be embedded in an interactive system or the training data changes frequently, generation efficiency becomes important.

Due to the requirement for retrieval efficiency, a classifier requiring objects to be retrieved one at a time into memory from the database before the classification function can be applied to them is not appropriate for database mining applications. Neural nets (see [11] for a survey) fit in this category. A neural net is a fixed sized data structure with the output of one node feeding into one or many other nodes. The classification functions generated by neural nets are buried in the weights on the inter-node links. Even articulating these functions is a problem, let alone using them for efficient retrieval. Neural nets learn classification functions by multiple passes over the training set till the net converges, and have poor generation efficiency. Neural nets also do not handle non-numerical data well.

Another important family of classifiers is based on decision trees (see [7] [6] [12] for an overview). The basic idea behind tree classifiers is as follows[13]. Let $\mathcal{E}$ be a finite collection of objects. If $\mathcal{E}$ contains only objects of one group, the decision tree is just a leaf labeled with that group. Otherwise, let T be any test on an object with possible outcomes $O_1, O_2, \ldots, O_w$. Each object in $\mathcal{E}$ will give one of these outcomes for T, so T partitions $\mathcal{E}$ into $\{\mathcal{E}_1, \mathcal{E}_2, \ldots \mathcal{E}_w\}$ with $\mathcal{E}_i$ containing those objects having outcome O_i. If each $\mathcal{E}_i$ is replaced by a decision tree for $\mathcal{E}_i$, the result would be a decision tree for all of $\mathcal{E}$. As long as two or more $\mathcal{E}_i$'s are non-empty, each $\mathcal{E}_i$ is smaller than $\mathcal{E}$, and since $\mathcal{E}$ is finite, this procedure will terminate.

ID3 (and its variants such as C4.5) [13] [14] and CART [2] are the best-known examples of tree classifiers. These decision trees usually have a branch for every value of a non-numeric attribute at a decision node. A numeric attribute is handled by repeated *binary* decomposition of its range of values. The advantage of the binary decomposition is that it takes away the bias in favor of attributes with large number of values at the time of attribute selection. However, it has the disadvantage that it can lead to large decision trees, with unrelated attribute values being grouped together and with multiple tests for the same attribute [13]. Morever, binary decomposition may cause large increase in computation, since an attribute with w values has a

[1] The other major topic in classification is *unsupervised learning*. In unsupervised classification methods, clusters are first located in the feature space, and then the user decides which clusters represent which groups. See [9] for an overview of clustering algorithms.

computational requirement similar to $2^{w-1}-1$ binary attributes [13].

The essence of classification is to construct a decision tree that correctly classifies not only objects in the training set but also the unseen objects in the test data set[13]. An imperfect, smaller decision tree, rather than one that perfectly classifies all the known objects, usually is more accurate in classifying new objects because a decision tree that is perfect for the known objects may be overly sensitive to statistical idiosyncrasies of the given data set [2] [15]. To avoid overfitting the data, both ID3 and CART first obtain a large decision tree for the training set and then prune the tree (usually a large portion of it) starting from the leaves [2] [14] [15]. Developing the full tree and then pruning it leads to more accurate trees, but makes classifier generation expensive.

The interval classifier ($\mathcal{IC}$) we propose is also a tree classifier. It creates a branch for every value of a non-numeric attribute, but handles a numeric attribute by decomposing its range of values into k intervals. The value of k is algorithmically determined separately for each node. Thus, for numeric attributes, $\mathcal{IC}$ results in k-ary trees, and does not suffer from the disadvantages of the binary trees. $\mathcal{IC}$ does dynamic pruning as the tree is expanded to make the classifier generation phase efficient. By limiting tests at decision nodes to point and range predicates, $\mathcal{IC}$ generates decision trees that decompose the feature space into nested n-dimensional rectangular regions, each of which can be specified as a conjunction of point and range predicates. $\mathcal{IC}$ can, therefore, generate SQL queries for classification functions that can be optimized using the relational query optimizers and can exploit database indexes to realize retrieval efficiency.

The organization of the rest of the paper is as follows. In Section 2, we present the $\mathcal{IC}$ classifier generation algorithm. In Section 3, we present the results of the empirical evaluation of the performance of $\mathcal{IC}$. We consider the sensitivity of $\mathcal{IC}$ to various algorithm parameters and the noise in the training and test data. We also present results comparing $\mathcal{IC}$ to ID3. Besides presenting a classifier suitable for database mining applications, a secondary goal of this paper is to argue that database mining is an important research topic requiring attention from database perspective. In Section 4, we describe some new problems that arise from encapsulating the classification capability in database systems, which have not been considered in the classification literature. We also discuss extensions that will allow $\mathcal{IC}$ to be used in these new application domains. We conclude in Section 5.

2 $\mathcal{IC}$ Generation Algorithm

We assume for simplicity that the population database $\mathcal{D}$ consists of one relation. Such a relation can usually be obtained by appropriate joins. Each tuple of this relation has n attributes. Every tuple belongs to one of m groups in the population, but the group label is not known for the tuples in $\mathcal{D}$. We also have a training sample $\mathcal{E}$ of tuples. Tuples in $\mathcal{E}$ are structurally identical to tuples in $\mathcal{D}$, except that the training tuples have an additional attribute specifying their group label. Attributes can be categorical or non-categorical. Categorical attributes are those for which there are a finite discrete set of possible values. The number of possible values is usually small and have no natural ordering to allow interpolation between two values. Examples of categorical attributes include "make of car", "zip code", etc. Other attributes are non-categorical. Examples of non-categorical attributes include "salary", "age", etc.

We define an *interval* to be a range of values for a non-categorical attribute or a single value for a categorical attribute. Tuples having values for an attribute falling in an interval are said to *belong* to that interval. Each group can be assigned a count of the tuples belonging to an interval of an attribute with that group as the label. The function **winner_grp** uses the group counts to determine the *winning* group for an interval. A function called **winner_strength** categorizes each winning group as a *strong winner* or a *weak winner*. The corresponding interval is then called a *strong interval* or a *weak interval*.

$\mathcal{IC}$ generation consists of two main steps. The function **make_tree** creates the decision tree, leaves of which are labeled with one group label. A tree traversal algorithm then generates a classification function for each group by starting from the root and finding all paths to a particular group at the leaves. Each path gives rise to a conjunction of terms, each term being a point predicate or range predicate. Disjunction of these conjunctions, one corresponding to each path for a group, yields the classification function for the group. We will only describe the function **make_tree** here; the generation of classification functions from the decision tree is fairly straightforward.

The function **make_tree** has a recursive structure. It works on an interval (or subdomain) of an attribute. Initially, it is given the entire domain of each attribute. One of the attributes is selected to be the winner attribute in the classification predicate (see **next_attr**). A *goodness function* is used for this determination. It then uses the tuples belonging to the input subdomain to partition the domain of the winner attribute into strong and weak intervals (see **make_intervals**). Decisions regarding the winning group are made final for the strong intervals of the winner attribute. The function then recursively processes the weak intervals of the winner attribute. The function terminates when a stopping criteria is satisfied.

For ease of exposition, in Figure 1, we present a version of the function **make_tree** in which several details have been omitted. In the pseudo code below, we present a more detailed version of the function **make_tree**.

```
// Determine the best attribute to use next
// for classification
func next_attr(H: Histograms) returns Attr
{
 For every attribute A do {
  Compute the value of the
  goodness function for A
 }

 Let winner_attr = attr with the largest
  value for the goodness function
  // Example of a goodness function is
  // the information gain; see Remarks

 Return winner_attr
}

// Partition the domain of attribute into
// intervals.
proc make_intervals(A: Attribute,
                    H: Histograms)
{
 For each value v in histogram of A {
  // Determine winning group for value v
  // using histograms in H
  // Example: return the group that has
  // the largest frequency for the value v
  winner = winner_grp(H, A, v)

  // Determine if winner is strong or weak
  // Example: return strong if the ratio of
  // the frequency of the winning group to
  // the total frequency for the value v
  // is greater than a specified threshold
  strength =
    winner_strength(winner, H, A, v)

  Save the winner and strength information
   for value v of attribute A
 }

 If the domain of A can be ordered {
  Form intervals of domain values
   by merging adjacent values that have
   the same winner with the same strength
 }
 else {
  Each value forms an interval by itself
   // i.e., the left and right endpoints
   // of the interval are the same
 }
}

// Procedure to build classification tree.
// Called as make_tree(training_set)
function make_tree(tuples: Tupleset)
returns TreeNode
{
 If stopping criteria is satisfied
  // see Remarks below
  return NULL

 Create a new tree node N
 For each group G and attribute A do
  make_histogram(G, A, tuples)

 For every non-categorical attribute do
  Smooth the corresponding histograms
  // see Remarks for smoothing procedure

 Let H be the resultant set of histograms
  for all attributes
 winner_attr = next_attr(H)
 make_intervals(winner_attr, H)

 Save in N the winner_attr and also
  the strong and weak intervals
  corresponding to the winner_attr
  for all groups

 For each weak interval WI of
  the winner_attr do {
   remaining_tups =
    training set tuples satisfying
    the predicate for WI
   child of WI = make_tree(remaining_tups)
 }

 return N
}
```

procedure make_tree (tupleset *T*)

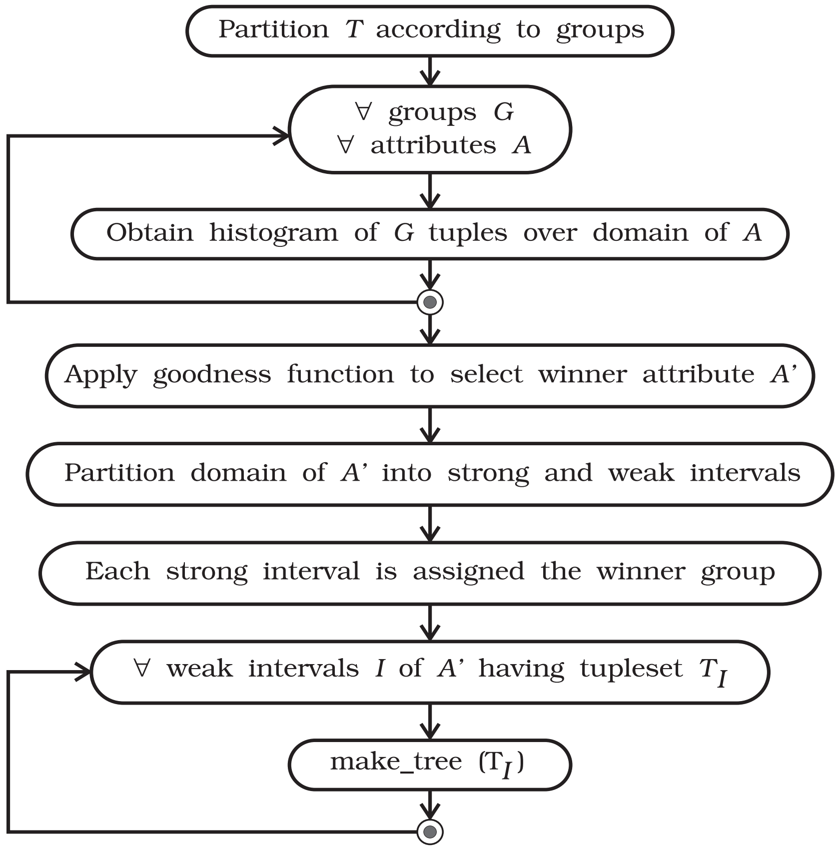

Figure 1: Procedure **make_tree**

REMARKS:
In the above description of the $\mathcal{IC}$ generation algorithm, we did not specify bodies of some of the functions. Our intention was to present a generic algorithm from which a whole family of algorithms may be obtained by instantiating these functions with different decision modules.
We now discuss the specific functions used in our implementation and also suggest some alternatives.
winner_grp: The function *winner_grp(H, attr, v)* returns the group that has the largest frequency for the value v of the attribute *attr* in histograms H. It is possible to use weighting if it is desired to bias the selection in favor of some specific groups.
next_attr: The function *next_attr(H)* is greedy, and selects the next branching attribute by considering one attribute at a time. (The problem of computing optimal decision trees has been shown to be NP-complete [8].) We consider two goodness functions: one minimizes the *resubstitution error rate*, the other maximizes the *information gain ratio*. Other possibilities for the goodness function include the cost of evaluating a predicate.
The *resubstitution error rate* [2] for an attribute is computed as

$$1 - \sum_v \text{winner\_freq}(v)/\text{total\_freq}$$

where *winner_freq(v)* is the frequency of the winning group for the attribute value v, and *total_freq* is the total frequency of all groups over all values of this attribute in histograms H.
The *information gain ratio* is an information theoretic measure proposed in [13]. Let the example set $\mathcal{E}$ of e objects contain e_k objects of group G_k. Then the entropy E of $\mathcal{E}$ is given by

$$E = -\sum_k \frac{e_k}{e} \log_2 \frac{e_k}{e}$$

If attribute A_i with values $\{a_i^1, a_i^2, \ldots, a_i^w\}$ is used as the branching attribute, it will partition $\mathcal{E}$ into $\{\mathcal{E}_i^1, \mathcal{E}_i^2, \ldots, \mathcal{E}_i^w\}$ with $\mathcal{E}_i^j$ containing e_i^j objects of $\mathcal{E}$ that have value a_i^j of A_i. If the expected entropy for the subtree of $\mathcal{E}_i^j$ is E_i^j, then the expected entropy for the tree with A_i as the root is the weighted average

$$E_i = \sum_j \frac{e_i^j}{e} E_i^j$$

The information gain by branching on A_i is therefore

$$\mathrm{gain}(A_i) = E - E_i$$

Now, the information content of the value of the attribute A_i can be expressed as

$$I(A_i) = -\sum_j \frac{e_i^j}{e} \log_2 \frac{e_i^j}{e}$$

The information gain ratio for attribute A_i is then defined to be the ratio

$$\mathrm{gain}(A_i)/I(A_i)$$

If two attributes are equally attractive, ties are currently broken by arbitrarily picking one of them. One could use additional criteria, such as the length of description, selectivity of attributes, etc. to break the ties.

winner_strength(winner, H, attr, v): The function *winner_strength* returns the strength as *strong* if the ratio of the frequency of the winning group *winner* to the total frequency for the value v of the attribute *attr* in H is above a certain *precision threshold*. Again, other criteria may be used to determine the strength. For example, besides precision threshold, one may require that there be certain minimum frequency at the value v before the winner is classified as strong.

The precision threshold may have a fixed value. For example, a fixed precision threshold of 1 has the effect that a winner is declared strong if instances of only the winning group are present. The precision threshold can also be an adaptive function of the current depth of the classification tree. The adaptive precision threshold we use is given by

$$1 - (\mathrm{curr\_depth}/\mathrm{max\_depth})^2$$

This function is conservative in the beginning in declaring a winner strong, but loosens the criteria as the tree grows.

Smoothing: Conceptually, we handle a non-categorical attribute by first generating a smooth frequency distribution from the histogram of its values. This distribution is then sampled at equi-distant points in its range of values. The number of sampling points is given by

max(minimum sampling points, sampling multiplier $\times$ number of distinct values in the histogram)

where *minimum sampling points* and the *sampling multiplier* are algorithm parameters. The smooth frequency distribution is not generated in practice. Rather, the frequency at a sampling point is determined using the following procedure.

Given a histogram $\{(v_i, f_i)\}$ of values of a non-categorical attribute, where f_i is the frequency of the attribute for value v_i, we need a way to interpolate for frequency for values not present in the histogram. Following the technique in [5], the frequency f for a value v is determined by considering the contribution of all values that occur in the histogram within an interval of length h centered at v. One way to interpret this method is to think of frequency f_i of every sample value v_i as being smeared over the interval $v_i - h/2$ to $v_i + h/2$ according to a weight function. If we let $W(u)$ be the "boxcar" weight function:

$$W(u) = \begin{cases} 1 & \text{if } \mathrm{abs}(u) < 1/2 \\ 0 & \text{otherwise} \end{cases}$$

then smearing can be thought of as replacing every the frequency f_i at value v_i by the function

$$f_i \times W((v - v_i)/h)/h$$

The total area under the curve $f_i \times W((v - v_i)/h)/h$ is f_i. The smoothed frequency f at any value v then is the sum of smears from all sample values:

$$f = 1/n \sum_{i=1}^{n} f_i \times W((v - v_i)/h)/h$$

Instead of the "boxcar" function, we use the raised cosine arch function:

$$W(u) = \begin{cases} 1 + 2\cos(2\pi u) & \text{if } \mathrm{abs}(u) < 1/2 \\ 0 & \text{otherwise} \end{cases}$$

This function decreases gradually from 0 to 1/2 and symmetrically from 0 to -1/2. Note that the area under the raised cosine arch is 1 and that the influence distance h represents a trade-off between smoothing and locality in terms of how much we want to constrain the smearing of frequency.

Stopping condition: Further branching from a node does not take place if all the intervals for the corresponding attribute are found to be strong. Similarly, if there are no tuples (or less than a specified number of tuples) in some range of values for the selected attribute, the corresponding interval is not further expanded. The winning group of the parent node is made the winner group in this empty interval. Branching may also be limited by specifying a maximum tree depth, in which case all weak intervals in a leaf node are treated as strong.

$\mathcal{IC}$ also provides a dynamic tree pruning criteria. For each node, an *expansion merit* is computed, and a node is expanded only if this merit is above an acceptable level. The expansion merit is based on the ideas in [14]. Suppose a tree T has been generated using N cases in the training set. Let some leaf account for K of these cases with J of them misclassified. The ratio J/K does not provide a reliable estimate of the error rate of the leaf for unseen cases, since the tree has been tailored to the training set. A more realistic error rate is obtained by applying continuity correction for the binomial distribution [19] in which J is repalaced by $J+1/2$. Let S be a subtree of T containing $L(S)$ leaves and let $\sum J$ and $\sum K$ be the corresponding sums over the leaves of S. S will misclassify $\sum J + L(S)/2$ out of $\sum K$ unseen cases. Now let E be the number of number of cases from the training set that a node misclassifies. This node is expanded only if $E + 1/2$ is beyond one standard error of $\sum J + L(S)/2$.

To avoid pruning too aggressively, $\mathcal{IC}$ also supports a lookahead procedure. The basic idea is that each node inherits from its parent a certain number of lookahead credits. If an expansion of a node does not result in acceptable error reduction, the node is still expanded if it still has credits left. Such a node passes one less credit to its children. If an acceptable level of error reduction takes place at a node, its credits are reset to the maximum. Further expansion does not take place if a node does not have any credit left and the amount of error reduction is not acceptable.

2.1 Example

We illustrate the $\mathcal{IC}$ classifier generation with a simple example. Consider a people database in which every tuple has only three attributes:

- age (**age**) – non-categorical attribute – uniformly distributed from 20 to 80
- the zip code of the town the person lives in (**zipcode**) – categorical attribute – uniformly distributed between 9 available zipcodes
- level of education (**elevel**) – categorical attribute – uniformly distributed from 0 to 4

Group membership depends only on **age** and **elevel**, and is independent of **zipcode**. There are only two groups in the population:

$$\begin{array}{lll} \text{Grp A:} & ((\mathbf{age} < 40) & \wedge \ (\mathbf{elevel} \in [0..1])) \vee \\ & ((40 \le \mathbf{age} < 60) & \wedge \ (\mathbf{elevel} \in [0..3])) \vee \\ & ((60 \ge \mathbf{age}) & \wedge \ ((\mathbf{elevel} = 0))) \\ \text{Grp B:} & \text{otherwise} & \end{array}$$

where ($\mathbf{elevel} \in [1..k]$) is equivalent to (($\mathbf{elevel} = 1) \vee (\mathbf{elevel} = 2) \vee \ldots \vee (\mathbf{elevel} = k)$). We have a training set of 1000 tuples that satisfy the above predicates.

$\mathcal{IC}$ first generates three histograms, one each for values in **age**, **zipcode**, and **elevel**. It smooths the histogram for the non-categorical attribute **age**, and chooses 100 equi-distant sampling values (an algorithm parameter) from the range of values for **age**. For each attribute value, it finds the winning group using the function **winner_grp**. The winning group for an attribute value is simply the group that has the largest frequency for that attribute value. We assume that the next attribute selection is based on the minimization of resubstitution error rate. The resubstitution error rate for an attribute is determined by adding for every value of the attribute the frequency of the winning group, dividing this sum by the total frequency, and subtracting this ratio from 1. The following are the values obtained for the resubstitution error rate for the three attributes:

Attribute	Error Rate
age	.227306
zipcode	.432000
elevel	.254000

Therefore, the function **next_attr** selects **age**, which has the minimum resubstitution error rate, as the next branching attribute.

$\mathcal{IC}$ uses the function **winner_strength** for each sample value of **age** to determine whether the winning group is strong or weak for that value. For a winner to be strong, the ratio of the frequency of the winning group to the total frequency at that value should be at least equal to a precision threshold. We used adaptive precision to dynamically adjust this threshold. For level 0 node, the value of this threshold is 1. For any of the values of **age**, the winner group is not found to be strong.

The function **make_interval** partitions the domain of **age** into intervals by merging adjacent values that have the same winner. The following three intervals are formed:

Attribute: **age**		
Interval	Winner	Strength
[20.00, 39.59)	Group B	Weak
[39.59, 59.79)	Group A	Weak
[59.79, 80.61)	Group A	Weak

Since all the three intervals are weak, the tree is further developed for them. $\mathcal{IC}$ partitions the original training set into three sets corresponding to the above three range of values for **age**, and the algorithm repeats for each set of training tuples. Let us consider the processing for the first set.

First, histograms of attribute values are developed for the tuples belonging to the reduced set. The resubstitution error rates are computed for the selection of the next attribute and the following values are obtained:

Attribute	Error Rate
age	.024345
zipcode	.340176
elevel	0

Therefore, **elevel** is selected as the next branching attribute. The precision threshold is reduced to 0.75 by the adaptive precision algorithm, and **winner_strength** finds the winning group to be strong for every value of **elevel**. Since **elevel** is a categorical attribute, each of its values is considered to be an interval and we have the following intervals:

Attribute: **elevel**		
Interval	Winner	Strength
[0]	Group A	Strong
[1]	Group A	Strong
[2]	Group B	Strong
[3]	Group B	Strong
[4]	Group B	Strong

Since all the intervals are strong, the tree is not grown further. If all or some of the intervals were weak, the algorithm would develop the tree further for those intervals. Figure 2 shows the decision tree generated by $\mathcal{IC}$. It is a coincidence that the next attribute

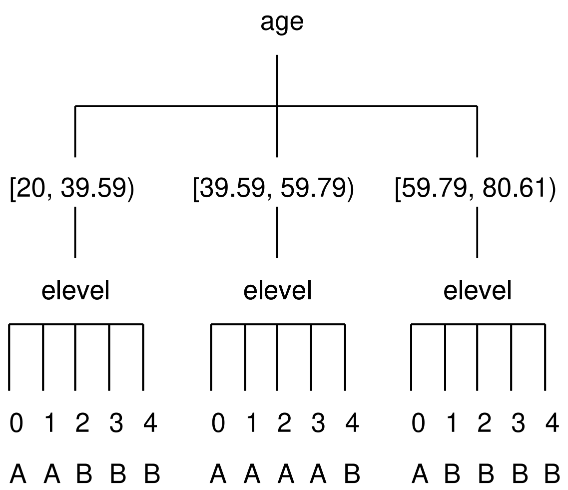

Figure 2: Example Decision Tree Generated by $\mathcal{IC}$

selected for each of the three intervals of **age** turned out to be **elevel**. In general, the siblings may not be the same attribute and different attributes may be selected for each of the intervals. Also, the tree need not be balanced — different branches could grow to different depth.

Thus, $\mathcal{IC}$ infers the following classification functions

$$
\begin{array}{ll}
\text{Grp A:} & ((20 \leq \textbf{age} < 39.6) \;\wedge\; (\textbf{elevel} \in [0..1])) \vee \\
& ((39.6 \leq \textbf{age} < 59.8) \;\wedge\; (\textbf{elevel} \in [0..3])) \vee \\
& ((59.8 \leq \textbf{age} < 80.6) \;\wedge\; ((\textbf{elevel} = 0)))
\end{array}
$$

$$
\begin{array}{ll}
\text{Grp B:} & ((20 \leq \textbf{age} < 39.6) \;\wedge\; (\textbf{elevel} \in [2..4])) \vee \\
& ((39.6 \leq \textbf{age} < 59.8) \;\wedge\; (\textbf{elevel} = 4)) \vee \\
& ((59.8 \leq \textbf{age} < 80.6) \;\wedge\; (\textbf{elevel} \in [1..4]))
\end{array}
$$

which is very close to the original rules. Note that the actual age range in the training set was from 20 to 80. It is easy for someone familiar with SQL to see how $\mathcal{IC}$ can generate SQL queries from the decision tree it synthesizes.

Let us remark on two termination conditions that are not illustrated in the above example. Firstly, suppose we had limited the classifier depth to 1. Since the algorithm treats all weak intervals in the leaf node as strong, we would have obtained the following classification functions:

$$\text{Grp A:} \;\; ((39.6 \leq \textbf{age} < 59.8))$$

$$\text{Grp B:} \;\; ((20 \leq \textbf{age} < 39.6) \;\vee\; (59.8 \leq \textbf{age} < 80.6))$$

The other case arises when there are less than a certain specified number of tuples in some range of values for the selected attribute. For example, having selected **elevel** as the next attribute as above, we may find that there is no tuple for **elevel** $= 0$. Then, the winning group of the parent node is made the winner group in this empty interval.

3 Performance

The goodness of a classifier has several dimensions:

1. *Generation Efficiency*: How efficient is the classifier generation process.
2. *Retrieval Efficiency*: How efficient is the classifier in retrieving all instances of a specified group.
3. *Classification Accuracy*: How correct is the classification of instances into groups.

Compared to ID3 and CART, the generation efficiency in $\mathcal{IC}$ stems from doing k-ary decomposition, instead of binary decomposition, of the range of a non-categorical attribute, and from using dynamic pruning, rather than back-tracking. Since $\mathcal{IC}$ develops k-ary trees, instead of binary trees, the trees in $\mathcal{IC}$ should be smaller and shallower. This should result in smaller queries, leading to better retrieval performance.

The *classification error*, that is, the fraction of instances in the test data that are incorrectly classified, is the classical measure of the quality of a classifier. We performed several experiments to empirically determine the accuracy of the classification functions generated by $\mathcal{IC}$. We first describe the experimental set up and the evaluation methodology, and then present some results, including comparison with ID3.

3.1 Methodology

We developed synthetic data to empirically evaluate the classification errors produced by $\mathcal{IC}$. The synthetic data is for a person database in which each person has the nine attributes given in Table 1. Attributes **elevel**, **car**, and **zipcode** are categorical attributes, all other are non-categorical attributes. Attribute values were randomly generated. There is also a derived attribute called **equity**, defined as follows:

$$\textbf{hyears} < 20 \Longrightarrow 0$$
$$\textbf{hyears} \geq 20 \Longrightarrow 0.1 \times \textbf{hvalue} \times (\textbf{hyears} - 20)$$

We developed a series of classification functions of increasing complexity that used the above attributes to classify people into different groups. There are 5 functions, labeled 1 through 5, involving 2 groups. Functions 1, 2, and 3 involve predicates with ranges on one, two, and three attribute values respectively. Function 4 is a linear function and Function 5 is a non-linear function of attribute values. These functions are listed in Appendix A.

For every experiment, we generated a training set and a test data set. Tuples in the training set were assigned the group label by first generating the tuple and then applying the classification function on the tuple to determine the group to which the tuple belongs. Labels were also generated for tuples in the test data set as per the classification function to determine whether the classifier correctly identified the group for the tuple or not.

It is rarely the case that the boundaries between the groups are very sharp. To model fuzzy boundaries, the data generation program takes a perturbation factor p as an additional argument. After determining the values of different attributes of a tuple and assigning it a group label, the values for non-categorical attributes are perturbed. If the value of an attribute A_i for a tuple t is v and the range of values of A_i is a, then the value of A_i for t after perturbation becomes $v+r\times p\times a$, where r is a uniform random variable between -0.5 and +0.5.

For each experimental run, the errors for all the groups are summed to obtain the classification error. For each classification function, 100 replications were done with new training sets being generated. The replications were then used to calculate the mean error with 95% confidence intervals. Errors are reported as percentages of the total test data set. In cases where the test data was perturbed, the intrinsic error in the test data was subtracted from the total error to arrive at the error due to misclassification.

We used training sets of 2500 tuples and test data sets of 10000 tuples. Before settling on these sizes, we studied the sensitivity of $\mathcal{IC}$ to these sizes. The training set was reduced from 2500 tuples to 1000 tuples in steps of 500. As expected, the classification error increased with decreasing training set size, but the increase in mean error was small. In database mining applications involving databases in gigabytes, the training sets are likely to be fairly large, and training sets of 2500 tuples are not unreasonable. We increased the test data sizes from 10000 to 25000, 50000, and 100000 tuples. The results indicated that 10000 tuples provided almost identical error estimates as larger test data sets, and we decided to stay with 10000 tuple test data sets to conserve computing time.

3.2 Classifier Accuracy

The first set of experimental results presented in Figure 3 show the classification error rates for the five functions. The results have been shown for the following versions of $\mathcal{IC}$:

1. *Error Pruning*: This version used pruning based on error reduction with lookahead described in Section 2. A lookahead of 5 was used, and no interval was declared strong unless all the tuples in that interval belonged to the winning group (i.e., precision threshold = 1.0).

2. *Adaptive Precision*: In this version adaptive precision threshold is used for deciding winner strength. A maximum depth of 10 was used to limit the growth of the tree.

3. *Fixed Precision*: This version used a fixed precision threshold of 0.9. A maximum depth of 10 was also used in this version.

All the three versions used *information gain ratio* as the criteria for the next attribute selection. The values of all the non-categorical attributes were perturbed by 5% for all the tuples in the training set and the test data set. The minimum sampling values for the non-categorical attributes were 100 and the sampling multiplier of 0.10 was used.

The first three functions partition the attribute space into hyper-parallelopipeds (n-dimensional rectangular regions). $\mathcal{IC}$ works very well for such classification

Attribute	Description	Value
salary	salary	uniformly distributed from 20000 to 150000
commission	commission	**salary** $>=$ 75000 $\Longrightarrow$ **commission** $=$ 0 else uniformly distributed from 10000 to 75000
age	age	uniformly distributed from 20 to 80
elevel	education level	uniformly chosen from 0 to 4
car	make of the car	uniformly chosen from 1 to 20
zipcode	zip code of the town	uniformly chosen from 9 available zipcodes
hvalue	value of the house	uniformly distributed from $0.5k100000$ to $1.5k100000$ where $k \in \{0 \cdots 9\}$ depends on **zipcode**
hyears	years house owned	uniformly distributed from 1 to 30
loan	total loan amount	uniformly distributed from 0 to 500000

Table 1: Description of Attributes

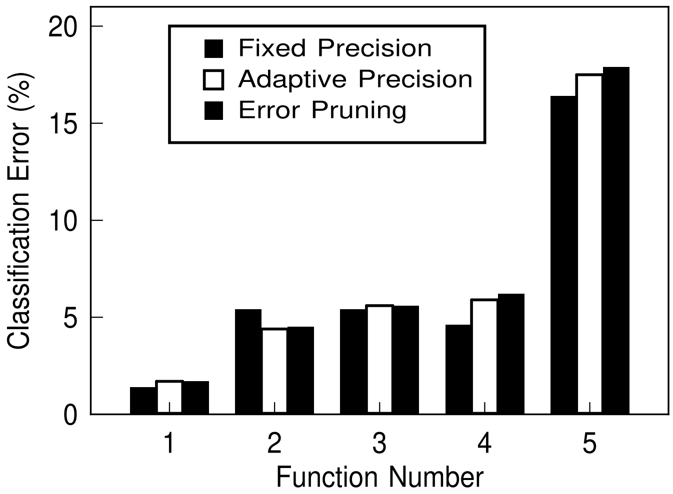

Figure 3: Comparing 3 Versions of $\mathcal{IC}$

functions. Functions 4 and 5 partition the attribute space into hyper-polyhedra. Now $\mathcal{IC}$ has to approximate the partitioning using n-dimensional rectangular regions. Hence the error is expected to increase.

The performance of the three versions of $\mathcal{IC}$ is pretty close. We were somewhat surprised by the superiority (albeit, only little) of fixed and adaptive precision versions over the error pruning version as they are much simpler and computationally much cheaper algorithms. Although we have presented the results only for perturbation of 5%, similar results were obtained for other perturbation values.

The following is a summary of the results from other sensitivity experiments (the constraint on number of pages prohibits us from presenting data from these experiments):

- The information gain ratio performed somewhat better than the resubstitution error rate as the criterion for the next attribute selection, but the difference was not large. The resubstitution error is a computationally cheaper metric.

- Increasing lookahead to 10 did not improve the performance of error pruning. A lookahead of 2 did as well as 5.

- Increasing the maximum depth to 15 did not help adaptive precision. The error rate with maximum depth of 5 were not very different from that of maximum depth of 10. It supports our conjecture that we may work with shallower trees when using $\mathcal{IC}$.

- The performance of $\mathcal{IC}$ seems to be sensitive to the smoothing parameters. We need to explore the parameter space further, and develop better understanding of it. Our conjecture is that we can improve the performance of $\mathcal{IC}$ by fine tuning the smoothing parameters.

3.3 Sensitivity to Noise

Insensitivity with respect to noise in the training and test data is an important quality of classifiers. Figure 4 shows the error rates for the Adaptive Precision $\mathcal{IC}$ for different amounts of perturbations in the data. These experiments were performed with the maximum depth of the tree set to 10.

The results show that $\mathcal{IC}$ is fairly stable. Errors increased as expected for Functions 1, 2, and 3, but only very moderately. We were surprised by reduction in error rates with increase in perturbation for Functions 4 and 5. But we found similar behavior with ID3. The

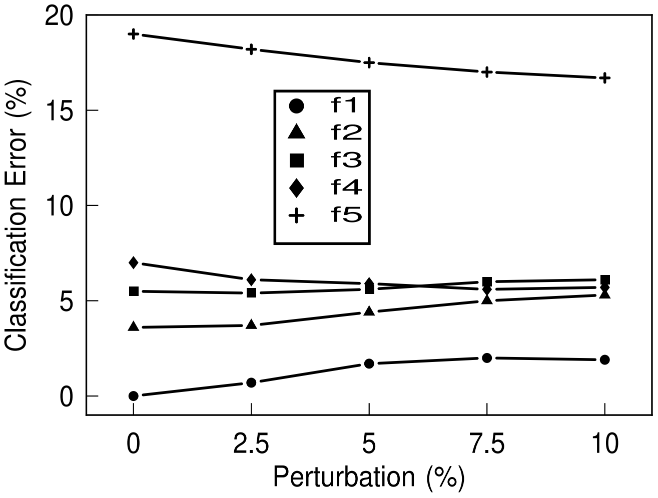

Figure 4: Sensitivity to Noise of Adaptive Precision

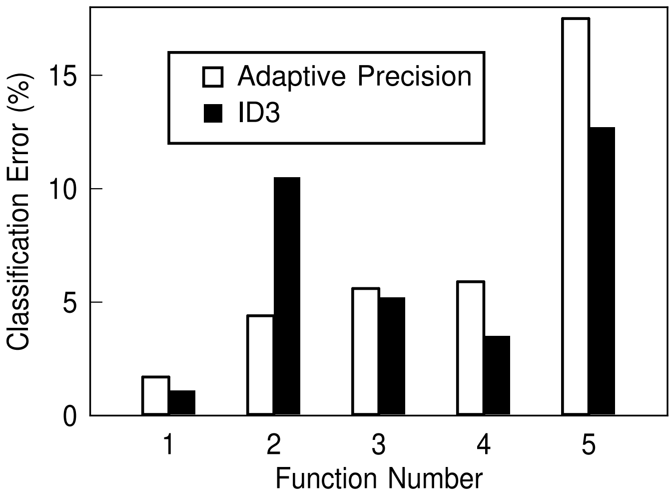

Figure 5: Comparing $\mathcal{IC}$ to ID3

explanation seems to be that since we subtract intrinsic error rate in the test data from the observed error rate to arrive at the error rate due to the classifier, some of the tuples that would have been reported as misclassified end up being in the intrinsic error pool, bringing the effective error rate down.

3.4 Comparison with ID3

We obtained the IND tree package [3] from the NASA Ames Research Center and ported it to IBM RISC System/6000 to compare the performance of $\mathcal{IC}$ with other classifiers. Experiments have been performed by the IND designers to ensure that IND reimplements C4 reasonably well. We present in this section the results of the comparison of the classification errors produced by ID3 (really C4) and $\mathcal{IC}$.

Figure 5 shows the comparative error rates for the Adaptive Precision $\mathcal{IC}$ and ID3 for 5% perturbation in training and test data. For Function 5, ID3 beats $\mathcal{IC}$ (12.5% vs. 17.5% average error rate), but $\mathcal{IC}$ beats ID3 for Function 2 (4.4% vs. 10.5% average error rate). The error rates for other functions are quite close, with ID3 doing a little better. A maximum depth of 10 was used for the adaptive Precision $\mathcal{IC}$. The difference in error rates did not change much between the two algorithms for different perturbation values (data not shown here), except that the performance of the two became identical for perturbation = 0 for Function 1, and for perturbation = 10% for Function 3. Given that $\mathcal{IC}$ does dynamic pruning to gain generation efficiency and ID3 fully expands the tree and then backtracks to prune it, the best we expected was that $\mathcal{IC}$ would come close to ID3 in the classification accuracy. Hence we feel satisfied with the classification accuracy shown by $\mathcal{IC}$. We also think that the error accuracy of $\mathcal{IC}$ can be improved by fine tuning the smoothing parameters.

4 New Applications

A secondary goal of this paper was to present the case that database mining applications lead to new interesting research problems. We now briefly summarize some of the new applications and research problems that we could identify in the context of our work on classification for database mining. These new applications call for integration of retrieval and classification components leading to a tight coupling of these functionalities. In fact, in our view, classification should be encapsulated as a function of database systems to meet these increasing demands. In order to do this, the generation, retrieval and classification times should be added as measures of quality of a classification method. We outline here three applications:

4.1 Best N problem

In target marketing, instead of requiring all individuals belonging to a group, very often the best N target individuals for a promotion are desired. The objective is to maximize profit, where profit is calculated as the difference between the amount of money made on all cumulative sales resulting from positive responses - cost of mailing - cost of retrieval of N candidates from the database. Due to the inclusion of retrieval cost, a solution with lower positive response may be selected over the one with the higher response, if the former has a much smaller retrieval cost.

The classification function generated by $\mathcal{IC}$ is a disjunction D of conjuncts C. Assuming that D selects more than N tuples, the problem is to determine those conjuncts in C whose disjunction selects at least N tuples and the profit is maximized at the same time. For each conjunct c in C, we can define two parameters:

1. The expected error of c, error(c) = 1 - return(c), where return(c) is the expected number of positive responses from those targets that satisfy c.

2. The expected retrieval cost of c, denoted by retrieve(c).

The profit from using c is given by

$$\begin{aligned}\mathrm{Profit}(c) &= \mathrm{return}(c) \times \mathrm{sale\_profit} - \\ &\quad (\mathrm{mail\_cost}(c) + \mathrm{retrieve}(c))\end{aligned}$$

where mail_cost(c) is the cost of mailing to the targets satisfying conjunct c.
In the worst case, the conjunct c can be retrieved by a sequential scan of the database, in which case the retrieval cost in number of I/Os will equal the number of blocks B that the database occupies. Assume that the database has indexes available on some attributes. Each indexed attribute A_i is characterized by an *adjusted selectivity*: $s(A_i)$, which is the average number of I/O (in blocks) necessary to access all records with $A_i = a$ where a ranges over all elements of the domain of A_i. In case A_i has the clustering index then $s(A_i)$ is equal to the selectivity of A_i divided by the number of records per block. If any indexed attribute appears in c, we may use the index to selectively retrieve records and then apply c to them. Assuming no index ANDing for simplicity, we can calculate the retrieval cost of c as

$$\min\{B, \min\{s(A_i) : A_i \text{ occurring in } c\}\}$$

Calculate the profit for each conjunct c in C. Sort conjuncts according to the value of the profit. Take first K conjuncts that together cover N targets, remembering that the total I/O cost for K conjuncts ceases to be additive after the retrieval cost exceeds the number of blocks in the database. In that case the retrieval cost will be constant and the final profit will depend only on the response rate of the first K conjuncts.
Interesting open questions involve the performance of the above method compared to a hypothetical "special purpose" classifier that makes attribute selections on the basis of the base profit gain for the Best N problem. We are currently working on this problem.

4.2 Adhoc Queries and Missing Data

Suppose that a market researcher would like to try a hypothetical new package which is similar to some of the packages used in the past. She would like to estimate its performance on a population which was not a target for the previous mailing. For example, she may decide to test a package which has both ski vacation and a 3-day tour of Paris. If she had some past data about customers who, in the past, took a ski package and those who took Paris vacation, she may decide to "compute the profile" for a union of the set of those customers and use it in estimating the (missing) values of the attribute corresponding to a new package and use this profile on the new population. After that she may change her mind and modify slightly the package, starting the next iteration. These successive iterations capture the "ad hoc", unexpected, nature of the planning process for a new marketing campaign.
For this scenario to become realistic, not only the expected retrieval time should be minimized but also the classifier generation time, since it contributes to the overall run time of the query. How do we decide what part of the classification task will be performed at compile time and which part at query run time?

4.3 Filters

A classifier that provides a rough classification but generates a profile that has "good" retrieval properties can be used as a "filter" for another classifier with good error characteristics, but poor retrieval properties. An example of a classifier with good error but poor retrieval is a neural net. A tree classifier, on the other hand could work as a filter, if it displays good retrieval performance. This involves, in general, building a classifier that best approximates a given "black box" function and has desirable retrieval characteristics.

5 Summary

We considered the problem of synthesizing classification functions for retrieving all instances of specified groups from a large database based on a representative sample of examples, and presented a tree-based interval classifier ($\mathcal{IC}$) for this purpose. The classifier is designed to be interfaced efficiently with the database systems. Since such classifiers may be embedded in interactive loops to answer adhoc queries about attributes with missing values, $\mathcal{IC}$ has been designed to be efficient in the generation of classification functions. The novel aspect of $\mathcal{IC}$ is its treatment of non-categorical attributes. Instead of creating binary subtrees for such attributes as is the case with ID3 and CART, $\mathcal{IC}$ creates k-ary subtrees, where k is algorithmically determined for each node. Rather than generating a full tree and pruning it, $\mathcal{IC}$ does dynamic pruning as the tree is expanded to make the classifier generation phase efficient. $\mathcal{IC}$ generates SQL queries for classification functions that can be optimized using relational query optimizers to realize retrieval efficiency. Preliminary empirical comparison with ID3 indicates that $\mathcal{IC}$ not only has retrieval efficiency and classifier generation efficiency advantages, but also compares quite favorably in the classification accuracy.

We also argued that classification should be encapsulated as part of future database systems. It then opens up new application areas for classifiers, not hitherto considered in the classification literature. We described extensions to $\mathcal{IC}$ for it to be used in these new application domains, and also presented some interesting open problems.

A by-product of this work has been the development of a systematic methodology for evaluating the performance of various classifiers. Our approach and the benchmarks we are developing allows one to systematically explore various operating regions. Our hope is that these benchmarks will serve the same role in classifier performance evaluation as the Wisconsin Benchmarks [1] played in the evaluation of relational query processing strategies.

The work reported in this paper has been done in the context of the Quest project at the IBM Almaden Research Center. In Quest, we are exploring the various aspects of the database mining problem. Besides classification, some other problems that we have looked into include the enhancement of the database capability with "what goes together" kinds of association queries and queries over large sequences such as stock tables. We believe that database mining is an important application area for databases and we hope that it will be developed into an important research topic by the database community.

6 Acknowledgments

We wish to thank Byron Dom and Wayne Niblack for technical discussions and information on current classifiers. Wolf-Ekkehard Blanz, Dragutin Petkovic, and David Steele provided useful pointers into the image processing literature. Byron Dom, Bruce Lindsay, Allen Luniewski, Eli Messinger, and Wayne Niblack provided insightful comments on an earlier version of this paper. We thank Wray Buntine for providing us the IND tree package that allowed us to compare $\mathcal{IC}$ with ID3. Finally, we wish to thank Irv Traiger for bringing the problem of database mining to our attention.

7 Appendix A

In the following, $(X \in [1..k])$ is equivalent to $((X = 1) \vee (X = 2) \vee \ldots \vee (X = k))$.

Function 1

$$
\begin{array}{ll}
\text{Grp A:} & ((\textbf{age} < 40) \vee ((60 \leq \textbf{age}) \\
\text{Grp B:} & \text{otherwise}
\end{array}
$$

Function 2

$$
\begin{array}{lll}
\text{Grp A:} & ((\textbf{age} < 40) & \wedge (50K \leq \textbf{salary} \leq 100K)) \vee \\
& ((40 \leq \textbf{age} < 60) & \wedge (75K \leq \textbf{salary} \geq 125K)) \vee \\
& ((\textbf{age} \geq 60) & \wedge (25K \leq \textbf{salary} \leq 75K)) \\
\text{Grp B:} & \text{otherwise} &
\end{array}
$$

Function 3

$$
\begin{array}{ll}
\text{Grp A:} & ((\textbf{age} < 40) \wedge \\
& \quad (((\textbf{elevel} \in [0..1]) \ \wedge \ (25K \leq \textbf{salary} \leq 75K)) \vee \\
& \quad ((\textbf{elevel} \in [2..3]) \ \wedge \ (50K \leq \textbf{salary} \leq 100K)))) \vee \\
& ((40 \leq \textbf{age} < 60) \wedge \\
& \quad (((\textbf{elevel} \in [1..3]) \ \wedge \ (50K \leq \textbf{salary} \leq 100K)) \vee \\
& \quad (((\textbf{elevel} = 4)) \ \wedge \ (75K \leq \textbf{salary} \leq 125K)))) \vee \\
& ((\textbf{age} \geq 60) \wedge \\
& \quad (((\textbf{elevel} \in [2..4]) \ \wedge \ (50K \leq \textbf{salary} \leq 100K)) \vee \\
& \quad (((\textbf{elevel} = 1)) \ \wedge \ (25K \leq \textbf{salary} \leq 75K)))) \\
\text{Grp B:} & \text{otherwise}
\end{array}
$$

Function 4

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{disposable} = (0.67 \times (\textbf{salary} + \textbf{commission}) - 0.2 \times \textbf{loan} - 10K} \\
\text{Grp A:} & \textbf{disposable} > 0 \\
\text{Grp B:} & \text{otherwise}
\end{array}
$$

Function 5

$$
\begin{array}{l}
\textbf{hyears} < 20 \Longrightarrow \textbf{equity} = 0 \\
\textbf{hyears} \geq 20 \Longrightarrow \textbf{equity} = 0.1 \times \textbf{hvalue} \times (\textbf{hyears} - 20)
\end{array}
$$

$$
\begin{array}{lll}
\textbf{disposable} & = & (0.67 \times (\textbf{salary} + \textbf{commission}) - \\
& & 0.2 \times \textbf{loan} + 0.2 \times \textbf{equity} - 10K) \\
\text{Grp A:} & & \textbf{disposable} > 0 \\
\text{Grp B:} & & \text{otherwise}
\end{array}
$$

References

[1] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill, "Benchmarking Database Systems: A Systematic Approach", *VLDB 83*, Florence, Italy, 1983.

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

[3] Wray Buntine, *About the IND Tree Package*, NASA Ames Research Center, Moffett Field, California, September 1991.

[4] Wray Buntine and Matha Del Alto (Editors), *Collected Notes on the Workshop for Pattern Discovery in Large Databases*, Technical Report FIA-91-07, NASA Ames Research Center, Moffett Field, California, April 1991.

[5] John M. Chambers, William S. Cleaveland, Beat Kleiner, and Paul A. Tukey, *Graphical Methods of Data Analysis*, Wadsworth International Group (Duxbury Press), 1983.

[6] Philip Andrew Chou, "Application of Information Theory to Pattern Recognition and the Design of Decision Trees and Trellises", Ph.D. Thesis, Stanford University, California, June 1988.

[7] G. R. Dattatreya and L. N. Kanal, "Decision Trees in Pattern Recognition", In *Progress in Pattern Recognition 2*, L. N. Kanal and A. Rosenfeld (Editors), Elsevier Science Publishers B.V. (North-Holland), 1985.

[8] L. Hayafil and R. L. Rivest, "Constructing Optimal Binary Decision Trees is NP-Complete", *Information Processing Letters*, **5**, 1, 1976, 15-17.

[9] A. K. Jain and R. C. Dube, *Algorithms for Clustering Data*, Prentice Hall, 1988.

[10] Ravi Krishnamurthy and Tomasz Imielinski, "Practitioner Problems in Need of Database Research: Research Directions in Knowledge Discovery", *SIGMOD Record*, Vol. 20, No. 3, Sept. 1991, 76-78.

[11] Richard P. Lippmann, "An Introduction to Computing with Neural Nets", *IEEE ASSP Magazine*, April 1987, 4-22.

[12] David J. Lubinsky, "Discovery from Databases: A Review of AI and Statistical Techniques", AT&T Bell Laboratories, Holmdel, New Jersey, June 1989.

[13] J. Ross Quinlan, "Induction of Decision Trees", *Machine Learning*, **1**, 1986, 81-106.

[14] J. Ross Quinlan, "Simplifying Decision Trees", *Int. J. Man-Machine Studies*, **27**, 1987, 221-234.

[15] J. Ross Quinlan and Ronald L. Rivest, "Inferring Decision Trees Using the Minimum Description Length Principle", *Information and Computation*, **80**, 1989, 227-248.

[16] G. Piatetsky-Shapiro and W. Frawley (Editors), *Proceedings of IJCAI-89 Workshop on Knowledge Discovery in Databases*, Detroit, Michigan, August 1989.

[17] G. Piatetsky-Shapiro (Editor), *Knowledge Discovery in Databases*, AAAI/MIT Press, 1991.

[18] G. Piatetsky-Shapiro (Editor), *Proceedings of AAAI-91 Workshop on Knowledge Discovery in Databases*, Anaheim, California, July 1991.

[19] G. W. Snedecor and W.G. Cochran, *Statistical Methods*, 7th Edition, Iowa State University Press, 1980.

[20] Shalom Tsur, "Data Dredging", *IEEE Data Engineering Bulletin*, **13**, 4, December 1990, 58-63.