On Discretization as a Preprocessing Step For Supervised Learning Models

A Thesis
Presented to the
Department of Computer Science
Brigham Young University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Dan Ventura
April 1995

This thesis, by Dan Ventura, is accepted in its present form by the Department of Computer Science of Brigham Young University as satisfying the thesis requirement for the degree of Master of Science.

____
Tony R. Martinez, Committee Chairman

____
William A. Barrett, Committee Member

____
J. Kelly Flanagan, Committee Member

__________ ______________________________
Date David W. Embley, Graduate Coordinator

Table of Contents

List of Figures, Tables, and Algorithms

Chapter 1
Introduction

Many machine learning and neurally inspired algorithms are limited, at least in their pure form, to working with nominal data, where nominal data is defined as data from a small, finite, unordered domain. Examples include CN2 [Cla89], ID3 [Qui90], AQ* [Mic90][Mic83], COBWEB [Fis90], and ASOCS [Mar91a][Mar91b][Mar90][Mar88][Mar86]. For many real-world problems, some provision must be made to support processing of continuously valued data. The main effort of this research is to investigate the feasibility of developing an effective and efficient method for discretization of continuously valued data as a preprocessing step for supervised learning systems. Discretization is the process of converting continuously valued data into nominal data by assigning ranges of real values to one nominal value (see Figure 1.1). Discretizing continuously valued data produces inherent generalization by grouping data into several ranges, representing it in a more general way. It has been shown that for some learning algorithms, efficient discretization as preprocessing resulted in significant speedup [Cat91]. Further, discretization has some psychological plausibility since in many cases humans apparently perform a similar preprocessing step representing temperature, weather, speed, etc., as nominal values. However, this thesis presents evidence that suggests *machine learning algorithms should directly incorporate handling of continuously valued data rather than relying on a discretization preprocessing step*.

It should be noted that from a pattern recognition and/or signal processing standpoint, this may be a somewhat unconventional use of the term discretization. In those disciplines, discretization refers to assigning a *discrete* value to a continuous one. For example, the real (continuous) value 93.39 might be discretized to 93 or 90 depending on the resolution of the A/D converter or discrete integer (or binary) representation available. In this context, discretization is a method for forcing “unrepresentable” values into the chosen representation. Discretization in this sense is simply a rounding off or truncation operation. For example, real-valued numbers are forced into 32-bit approximations so that computers can deal with them.

Discretization as used in this thesis, and in the machine learning literature in general, is an extension of this idea in which the representation that the continuously valued data will be discretized into need not be chosen in advance. Instead, a discretization method is expected to discover the best representation as well as the correct mapping between the continuously valued data and that representation. Thus discretization becomes a much more challenging problem. In the pattern recognition world, this is considered a classification problem and the representation is the set of classes discovered. However, in the context of machine learning and of this thesis, the term class refers to an output class and has no connection with the interval representation developed during discretization. During discretization, no classification is taking place.

These are subtle differences in usage and interpretation but they are important to note if the two fields of machine learning and pattern recognition are to coexist meaningfully.

Tropical	**Length**	Color	Type	
Yes	**7.5**	Blue	→	Poisonous
No	**1.2**	Red	→	Nonpoisonous
Yes	**4.5**	Green	→	Nonpoisonous
Yes	**9.2**	Green	→	Poisonous
Yes	**2.2**	Red	→	Nonpoisonous
		.		
		.		
		.		

↓

Discretizer

↓

Tropical	**Length**	Color	Type	
Yes	**Long**	Blue	→	Poisonous
No	**Short**	Red	→	Nonpoisonous
Yes	**Medium**	Green	→	Nonpoisonous
Yes	**Long**	Green	→	Poisonous
Yes	**Short**	Red	→	Nonpoisonous
		.		
		.		
		.		

Figure 1.1. Discretizing continuously valued data

1.1. Related Work

Many methods for general clustering of data suggest themselves for the purpose of discretizing or grouping continuously valued data. Methods considered include K-means, Mini-max, Bayes, and K-nearest neighbor

[Eve74][Jai88][Jar73][Spa80][Tou74]. These methods solve many general clustering problems. However, each of these methods has been empirically evaluated (except Bayes for the reason given below) on real-world applications, and none of them are particularly suited to this specific task of acting as a preprocessor for discretizing real data into nominal data. A short discussion of each follows.

1.1.1. *K-means.* K-means depends on user input for the number of desired classes, which is often unknown. Ideally, the "best" number of classes should be discovered through the discretization process. Also, K-means uses Euclidean distance as the metric for determining class membership. Typically, when considering each variable independently, this is not a reasonable metric because higher order correlations necessary for proper clustering are ignored. Also, there is no method for automatically determining which value of K is most appropriate.

1.1.2. *Maxi-min.* Maxi-min is somewhat the antithesis of K-means. It discovers a "best" number of clusters, depending on a user-defined threshold, which is usually problem dependent. In some cases combining Maxi-min with K-means improves performance, but again the metric for determining class membership is Euclidean distance, which is often not acceptable for the reason given above. Like K-means, there is no method for determining the best value for the threshold.

1.1.3. *Bayes.* Bayes guarantees minimization of error under the assumption of normally distributed classes (or the ability to functionally approximate the distribution) and knowledge of *a priori* probabilities. However, in discretizing data there initially exist no intervals for which to assume distributions and therefore no *a priori* probabilities for those intervals either. Bayes is a supervised clustering technique and therefore does not readily meet the requirements for automatic discretization, although some applications such as Cheeseman's AUTOCLASS do make use of Bayesian statistics in their classification schemes (see chapter 4). It is therefore more natural to compare the Bayesian approach with other supervised learning models (which models may or may not use discretization as a preprocessing step). Just such an empirical comparison is made in chapter 4.

1.1.4. *K-nearest neighbor.* K-nearest neighbor performs well if the user-defined parameters K (size of neighborhood) and T (number of shared neighbors) are properly defined. Unfortunately, "good" values for these parameters seem to be application dependent. Performance is also affected by initial ordering of data.

It is important to note that perhaps the main reason for the poor performance of the methods discussed above is due to the nature of the task (that of a preprocessor to a supervised learning system). In this situation, only first order correlations may be considered because the higher order correlations are to be explored by the supervised system. The methods discussed above depend on examining higher order correlations in order to make a classification and so in this context are not appropriate for consideration.

In the past, various methods using local minima-type approaches for detecting modes in first order data such as histograms have been used [Doy62][Gon87][Wes78][Whi83]. However, these suffer from the usual problems associated with local minima (i.e., Should the data be smoothed? If so, how much? What constitutes a local minimum?, etc.) As a result, these methods are often inadequate [Cho72]. As an alternative, statistical approaches that attempt to fit the data to a particular distribution have been proposed [Cho72]. However, these methods' statistical solutions to overcoming the problems associated with the local minima approaches are also their downfall whenever the data are such that they cannot be fitted to a particular known distribution.

Several methods specifically for discretization also exist including the equal-width-intervals and equal-frequency-intervals methods [Won87], Lebowitz's UNIMEM [Leb85], Catlett's D2 [Cat91], Schlimmer's extension to STAGGER [Sch87], Fayyad and Irani's extension to decision trees [Fay92], and Kerber's ChiMerge [Ker92]. These generally outperform the methods mentioned above since they are designed specifically for the task of discretization. However, these too have inherent weaknesses that limit their usefulness.

1.1.5. *Equal-width-* and *Equal-frequency-intervals* [Won87]. Equal-width-intervals is a generic method that simply divides the data into some number of intervals all with equal width. This is completely arbitrary and does not seek to discover any information inherent in the data. Its main advantage is its simplicity. Equal-frequency-intervals approaches the task from the opposite end, dividing the data into some number of intervals each of which contains the same number of data points. The same comments apply to this method as to the equal-width method.

1.1.6. *UNIMEM* [Leb85]. Lebowitz's UNIMEM system employs three different approaches to discretization: use of domain information, "looking for gaps", and generalization-based categorization. The first takes advantage of available domain-specific knowledge to help identify good cutpoints. For example, if a Federal law is effective only in those states with a legal drinking age of 21, then it may be expected that the age 21 would be a logical cutpoint in discretizing the age attribute. This type of information is used whenever possible, but in many cases such domain-specific knowledge may not exist. The second is basically a collection of number heuristics that employs the same approach that a teacher uses when grading on a curve--look for the natural breaks. Defining a break is the difficult part, and this is never explicitly addressed but is rather a function of subjective and somewhat arbitrary heuristics. The third method, generalization-based categorization, uses a testbed node that is a generalization (with respect to its nominal attributes) of several other sub-nodes. The testbed node examines its sub-nodes and looks for natural stratification of the attribute in question. This stratification is then used to discretize that attribute. This allows different discretizations of the same attribute for different testbed nodes (that is, different discretization of the same attribute for different concepts or output classes). However, this method assumes that at least some of the attributes are nominal to begin with so that generalization of the testbed node can take place. Some discussion is also given to the idea of changing discretizations over time (similar to the idea of concept drift) and to different discretizations for different contexts.

1.1.7. *D2* [Cat91]. Catlett presents a method called D2 whose main emphasis is the speeding up of some tree-based learning algorithms. For each attribute to be discretized, the following is applied. Choose the first cut-point as in ID3, that is, by maximizing gain. Next, instead of reapplying the gain criterion to choose the next attribute to split on, apply it to the same attribute to determine whether on not it should be split again. This continues recursively until one of four stopping conditions is met:

1. The number of examples in an interval is smaller than the system parameter *S*;
2. The number of intervals is larger than the system parameter *L*;
3. The gain for all intervals is equal;
4. All examples in the interval are in the same class.

Cases 1 and 2 prevent the creation of too many trivial intervals (*S* and *L* are system parameters) and cases 3 and 4 handle the situation when further splitting is unnecessary. D2 assumes two output classes so that problems with more than two classes must be considered as a set of binary classed problems where each class in turn is considered as the positive class with all others being considered as negative.

1.1.8. *STAGGER* [Sch87]. Schlimmer extends his STAGGER model to handle continuously valued attributes by using a statistical utility function that is applied to each potential cutpoint. The simplest approach would be to simply choose the cutpoint with the highest utility; however, in an effort to obtain improved results, a local smoothing function is first applied to the data and then all cutpoints that are locally maximal are selected. The utility function is calculated from all the examples seen so far and can be updated as new examples are observed. It is calculated as

$$U(endpoint) = \prod_{i=1}^{|classes|} \text{odds}(class_i) \times \frac{\text{p}(class_i \mid < endpoint)}{\text{p}(class_i \mid > endpoint)}.$$

1.1.9. *Decision Trees* [Fay92]. Fayyad and Irani present a method for discretization of continuously valued attributes in decision tree generation that is based on a measure of class entropy. Class entropy is defined as

$$\text{Ent}(S) = -\sum_{i=1}^{k} \text{P}(C_i, S) \log\big(\text{P}(C_i, S)\big),$$

where S is a set of examples, C_i is an output class, k is the number of output classes, and $\text{P}(C_i, S)$ is the proportion of examples in S that have class C_i. To discretize an attribute A, consider all possible cutpoints, T, calculating the *class information entropy induced by T*:

$$\text{E}(A, T; S) = \frac{|S_1|}{N}\text{Ent}(S_1) + \frac{|S_2|}{N}\text{Ent}(S_2),$$

where $N = |S|$, $S_1 = \{s \mid s \in S \text{ and } A\text{-value} < T\}$, and $S_2 = \{s \mid s \in S \text{ and } A\text{-value} \geq T\}$. The cutpoint for which $\text{E}(A,T;S)$ is minimal is chosen and this process may be applied recursively, with an appropriate stopping criterion to create more than two partitions of an attribute. They also provide a proof that cutpoints chosen in this manner are "good" in the sense that they always have values between two examples of different output classes.

1.1.10. *ChiMerge* [Ker92]. Chimerge is based on the statistical χ^2 test. All examples are sorted and initially each is placed in its own interval. Then intervals are merged iteratively in the following manner. The χ^2 value is computed for each pair of adjacent intervals and the pair of intervals with the lowest χ^2 value are merged. This process is repeated until all pairs of adjacent intervals have a χ^2 value that exceeds some threshold. Two adjacent intervals whose χ^2 value exceeds this threshold are considered to be significantly different and thus should not be merged. The χ^2 value for two adjacent intervals is calculated

$$\chi^2 = \sum_{i=1}^{m} \sum_{j=1}^{k} \frac{\left(A_{ij} - E_{ij}\right)^2}{E_{ij}}$$

where

$m = 2$ (the number of intervals being compared),
k = the number of output classes,
A_{ij} = the number of examples in the ith interval, jth class,
R_i = the number of examples in the ith interval = $\sum_{j=1}^{k} A_{ij}$,
C_j = the number of examples in the jth class = $\sum_{i=1}^{m} A_{ij}$,
N = the number of examples = $\sum_{j=1}^{k} C_j$,
E_{ij} = expected frequency of $A_{ij} = \dfrac{R_i \times C_j}{N}$.

The threshold is determined by selecting a significance level (usually somewhere between .9 and .99) and using a standard χ^2 table to look up the threshold value. To do this also requires knowing the degrees of freedom which will be one less than the number of output classes. For example, if there are 3 output classes and a significance of .9 is desired, a standard χ^2 table gives the threshold value as 4.6. This means that *if* the attribute under consideration and the output class are independent, then there is a 90% chance that $\chi^2 < 4.6$. Conversely, if $\chi^2 > 4.6$ this implies that the attribute and output class are not independent. ChiMerge is robust and makes use of *a priori* knowledge. However, it cannot discretize unsupervised data (data not represented as an instance with an output class). It performs only pairwise evaluation of neighboring intervals rather than evaluating the whole classification, and it depends on user-defined parameters.

Study of these methods has resulted in the compilation of an eclectic list, summarized in Table 1.2, of desirable attributes for a discretization method.

• Measure of interval set "goodness"
• No specific closeness measure
• No user-defined parameters
• Globality rather than locality
• Simplicity
• Use of feedback
• Use of *a priori* knowledge
• Higher order correlations
• Fast

Table 1.2. Desirable attributes of a discretization method

Chapter 2 presents an approach to discretization that attempts to incorporate the above attributes into a new and cohesive paradigm for general discretization as a preprocessing step. Chapter 3 then compares the methods presented in chapter 2 against several of the conventional methods discussed above. This comparison provides evidence that discretization as a preprocessing step may be improved upon if it is not considered as a completely autonomous step but rather as a more integral part of the supervised learning process. Therefore, chapter 4 presents work on attacking the problem from a different point of view, that of allowing the discretizer to perform as much learning as possible, yet still act in concert with other learning systems in many cases. Algorithms, analysis, and empirical results are included. Chapter 5 provides direction for future work and conclusions.

Chapter 2
BRACE: Boundary Ranking And Classification Evaluation

This chapter attempts to support models which naturally work with nominal data by presenting a method of quickly and intelligently discretizing continuously valued data into several useful ranges that can then be handled as nominal data. The method concentrates on finding the natural boundaries between ranges and creates a set of possible classifications using these boundaries. All classifications in the set are evaluated according to a criterion function and the classification that maximizes the criterion function is selected.

2.1. The BRACE Paradigm

Boundary Ranking And Clustering Evaluation (BRACE) is a paradigm designed to meet the objectives presented above. A high-level description of the paradigm consists of the following 4 steps:

1. Find all natural boundaries in the data;
2. Calculate the *rank* of each boundary;
3. Construct a set of possible classifications;
4. Evaluate all classifications in the set and choose the best one.

Each specific algorithmic instantiation of the BRACE paradigm uses a unique *boundary definition* (BD) for finding boundaries, *boundary ranking function* (BRF) for measuring each boundary and *classification optimization function* (COF) for evaluating classifications. A *classification* is defined as a grouping of the data into two or more discrete intervals. The definition and functions are based on some heuristic or bias and depend on the specific algorithm. One such algorithm is presented in section 2.2. Other algorithms that follow the BRACE paradigm may introduce other methods of finding boundaries in the data, measuring those boundaries, and evaluating the classifications in the set.

A set C of possible classifications is constructed which consists of the most likely 2-interval classification, the most likely 3-interval classification, etc. up to the most likely $B+1$-interval classification, where B is the number of boundaries identified. The first classification consists of dividing the data into two intervals by using the boundary with the highest rank as a division. The second classification is obtained by dividing the data into three intervals using the first two highest ranked boundaries. This continues until the data has been split into $B+1$ intervals and C contains B different classifications.

Once a set of possible classifications has been created, one of them must be selected for use in discretizing the data. Each of the classifications in the set are evaluated according to the COF and the one classification that maximizes it is selected. All values placed into the same range by the chosen classification are assigned the same nominal value.

2.2. VALLEY: An Instantiation of BRACE

Define:

H: as a histogram of the continuously valued data points. The *range* of the data is defined as the interval on the real numbers from the smallest data point to the largest. This range is broken up into an initial number of *intervals*, and the height of each interval is the number of data points that appear in it. It must be noted that the initial number of intervals may be considered a parameter. However, it is more a parameter of the physical implementation rather than a parameter of the logical algorithm. Any reasonably large number of intervals (~50) will reveal roughly the same profile for the data. Of course, the greater the number of intervals created, the greater the number of valleys that must be processed.
valley: as a local minimum in *H*.
V: as the set of all valleys.
rank(valley): as the rank (measure of "goodness") of a given valley.
C: as the set of possible classifications to be considered by the COF.
instance: as a set of attribute values (Left Hand Side) implying a set of output values (Right Hand Side). For example,

YES 7.5 BLUE --> TRUE

training set: as a set of instances. For example,

YES 7.5 BLUE --> TRUE
NO 1.2 RED --> FALSE
YES 4.5 GREEN --> FALSE
•
•
•

VALLEY: as an instantiation of the BRACE paradigm defined by its specific Boundary Definition, Boundary Ranking Function, and Classification Optimization Function:

BD (Boundary Definition): as a *valley* (local minimum) in a histogram. See Figure 2.1.

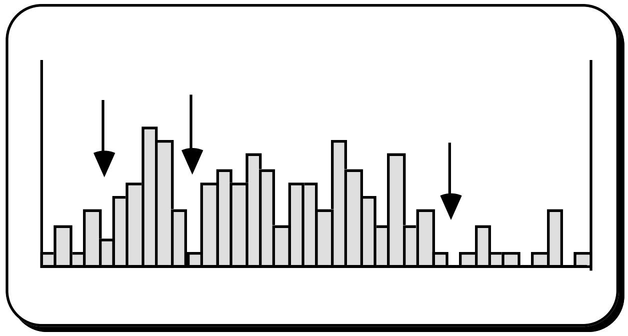

Figure 2.1. Arrows indicate several valleys

BRF (Boundary Ranking Function): as the area of a valley under the imaginary line between the valley's two bounding local maxima (figure 2.2.)

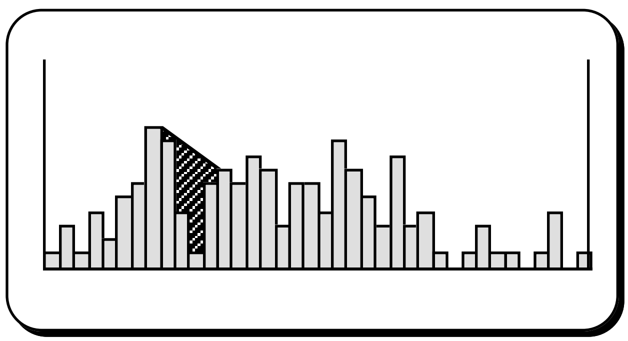

Figure 2.2. Area of a valley (shaded)

COF (Classification Optimization Function): as the number of consistent output class predictions minus the number of inconsistent output predictions. These predictions are made considering the discretization of the attribute currently being processed as the entire left-hand side of each instance. This definition depends on the data being part of an instance and is designed to minimize the *intra*class distance and maximize the *inter*set distance.

For example, consider the simple training set:

2.5	$\rightarrow$	Yes
5.3	$\rightarrow$	Yes
9.6	$\rightarrow$	No
1.9	$\rightarrow$	Yes
4.3	$\rightarrow$	No.

A discretization that categorizes values smaller than or equal to 5 as Small and those larger than 5 as Large would result in the modified training set:

Small	$\rightarrow$	Yes
Large	$\rightarrow$	Yes
Large	$\rightarrow$	No
Small	$\rightarrow$	Yes
Small	$\rightarrow$	No.

Since the value Small predicts two instances of Yes and one of No, and the value Large predicts one instance of Yes and one of No, the *COF* for this discretization is (2-1)+(1-1) = 1. However, a discretization that categorizes values 3 or less as Small, values 7 or less as Medium, and values greater than 7 as Large results in the modified training set:

Small	$\rightarrow$	Yes
Medium	$\rightarrow$	Yes
Large	$\rightarrow$	No
Small	$\rightarrow$	Yes
Medium	$\rightarrow$	No.

Now Small predicts two Yes, zero No, Medium predicts one of each and Large predicts a single No. Therefore, the *COF* is (2-0)+(1-1)+(1-0) = 3.

Inserting BD, BRF, and COF into BRACE results in the VALLEY algorithm, which consists of five basic steps:

1. Order the data and build histogram H.
2. Find the set V of all valleys.
3. For each valley $v \in V$, rank(v) = area of v.
4. Construct the set C of possible classifications.
5. Evaluate C and chose max{COF(c)}, $c \in C$.

The first three steps are relatively straightforward. Once the valley sizes for all valleys have been calculated, the set C of possible classifications is built using the biggest (in terms of area) valley, then the 2 biggest valleys and so on until we have split the histogram into $|V|+1$ intervals and obtained $|V|$ classifications in C (i.e. using BD and BRF.)

Since the data is being discretized for a *supervised* learning system, knowledge of the output value for each instance in the training set is assumed. It is important to note that this is an assumption made by the VALLEY algorithm and not by the general BRACE paradigm.

Using this knowledge, each classification in C is evaluated according to its ability to correctly predict output values for the training set, using only the current attribute under consideration. The number of inconsistent predictions is subtracted from the number of consistent predictions, and the result is the score for that classification. The classification with the highest score is selected (i.e. using COF.)

2.2.1. *An Example.* Consider the following *training set* of instances, only the first five of which are actually shown.

YES	7.5	BLUE	--> TRUE
NO	1.2	RED	--> FALSE
YES	4.5	GREEN	--> FALSE
YES	9.2	GREEN	--> TRUE
YES	2.2	RED	--> FALSE

•
•
•

Since the second attribute is continuously valued it must be discretized. Assume that H for attribute 2 ranges from 1 to 10 and looks as in Figure 2.3a. Finding all the valleys proceeds as in Figure 2.3b-d where the numbers above the various valleys indicate their relative rank.

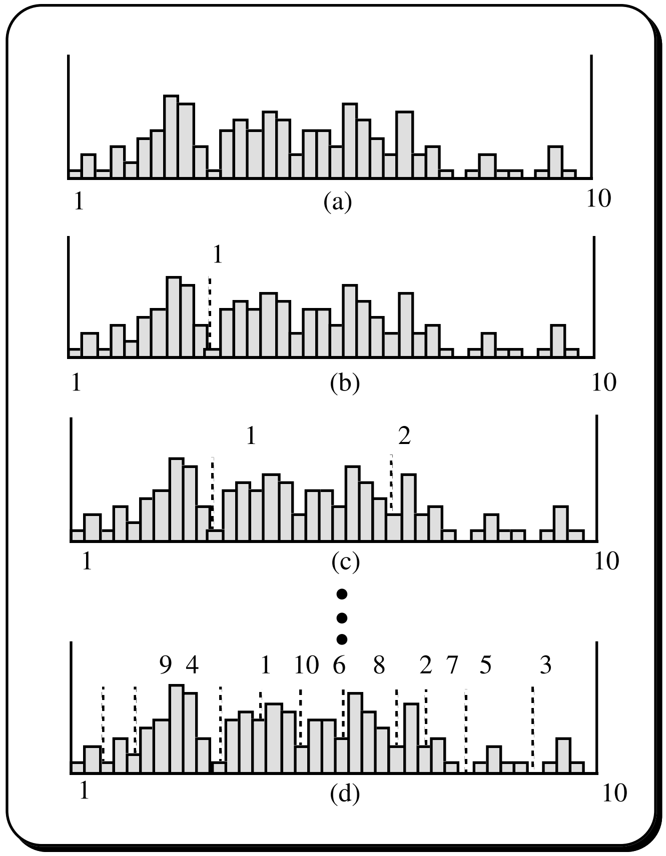

Figure 2.3. (a) Histogram H (b) 1 valley (c) 2 valleys (d) complete set V

Recall that the set C includes the classification shown in Figure 2.3b, the classification shown in Figure 2.3c, and all the classifications implied between steps c and d in Figure 2.3. Each of these classifications in C are evaluated as shown in Table 2.4. A simplified copy of the training set is "constructed" considering only attribute 2. The table is interpreted as follows: dividing the data into two intervals (as in Figure 2.3b), value 7.5 is in interval 2 and implies TRUE, value 1.2 is in interval 1 and implies FALSE, and so on. The score for column one is calculated by finding the number of consistent output predictions minus the number of inconsistent predictions.

2 int.	3 int.	4 int.	5 int.		11 int.
2→T	3→T	4→T	4→T		
1→F	1→F	1→F	1→F		
2→F	2→F	3→F	3→F	...	
2→T	3→T	4→T	5→T		
1→F	1→F	2→F	2→F		
		.			
		.			
		.			
3	5	5	5		

Table 2.4. Calculating scores for each classification in *C* using the COF

For the two interval case, interval 1 predicts F twice and interval 2 predicts T twice and F once. The total score for 2 intervals is therefore 4-1=3. The other columns of the table are calculated similarly. For the purpose of illustration, subscores calculated using only the 5 instances shown are used as the total column score.

Breaking the data into 3 intervals gives the maximum score. Note that every classification thereafter also gives the maximum score. This may not always be the case, however. Some training sets (i.e. those containing noise) may show slight improvement when the number of intervals is high enough. It is therefore necessary to find a balance between making the classification too general and learning random noise or insignificant variations in the training set. Therefore, in an atempt to do so and (hopefully) to facilitate generalization the smallest number of intervals with the maximum score is selected.

The data is discretized into three intervals as shown in Figure 2.3c. Values 1.2 and 2.2 are in class 1, 4.5 is in class 2, and 7.5 and 9.2 are in class 3. The modified instance set now looks like:

```
YES  3   BLUE    --> TRUE
NO   1   RED     --> FALSE
YES  2   GREEN   --> FALSE
YES  3   GREEN   --> TRUE
YES  1   RED     --> FALSE
           •
           •
           •
```

As a final comment, it should be noted that although the output in the preceding example is binary, VALLEY and BRACE can handle any nominal output. They are not limited to the binary case.

2.3. SLICE: An Extension of VALLEY

Of course a valley as defined above may not be the best or only natural division in the data. In fact, valleys may not be good divisions at all in some cases. Therefore, a generalization of VALLEY may be to consider all possible boundaries in a histogram H. This extension of the VALLEY algorithm is called SLICE and is defined below.
Define:

H: as above.
slice: as a division between intervals in *H*.
S: as the set of all slices.
C: as the set of possible classifications to be considered by the COF.
instance: as above.
training set: as above.
SLICE: as an instantiation of the BRACE paradigm defined by its specific Boundary Definition, Boundary Ranking Function, and Classification Optimization Function:
BD (Boundary Definition): is a *slice* in a histogram. See Figure 2.5.

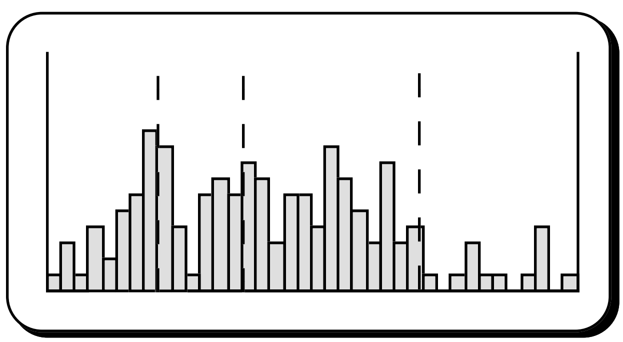

Figure 2.5. Dashed lines indicate several slices

BRF (Boundary Ranking Function): is not used in SLICE since all boundaries are considered to have equal rank.
COF (Classification Optimization Function): again depends on the data being part of an instance and is defined as above.

SLICE is a simple generalization of valley in which every division between intervals, instead of just local minima, is considered a candidate for splitting an interval. The main difference being that all *slices* are considered to have the same ranking so the BRF is unnecessary.

SLICE's operation is identical to VALLEY's. The data is sorted and used to create a histogram. Then for each continuously valued attribute, the data is classified using only that attribute as the lefthand side of the instance. All possible *slices* are then evaluated by temporarily discretizing the data using the *slice* and reclassifying the instances. If the *slice* with the best results does a better job of classification than no *slices*, that *slice* is made permanent and the process is repeated.

2.4. Empirical Results

VALLEY and SLICE were tested on various data sets from the UC Irvine machine learning database [Mur92]. One example is the hepatitis data set, which has a number of continuously valued variables with widely varying distributions. The results of discretizing these six continuously valued variables are displayed in Figures 2.6 and 2.7. Although some discussion follows on the performance of BRACE (in its VALLEY instantiation), the bulk of the empirical performance results and the resulting discussion is presented in chapter 3.

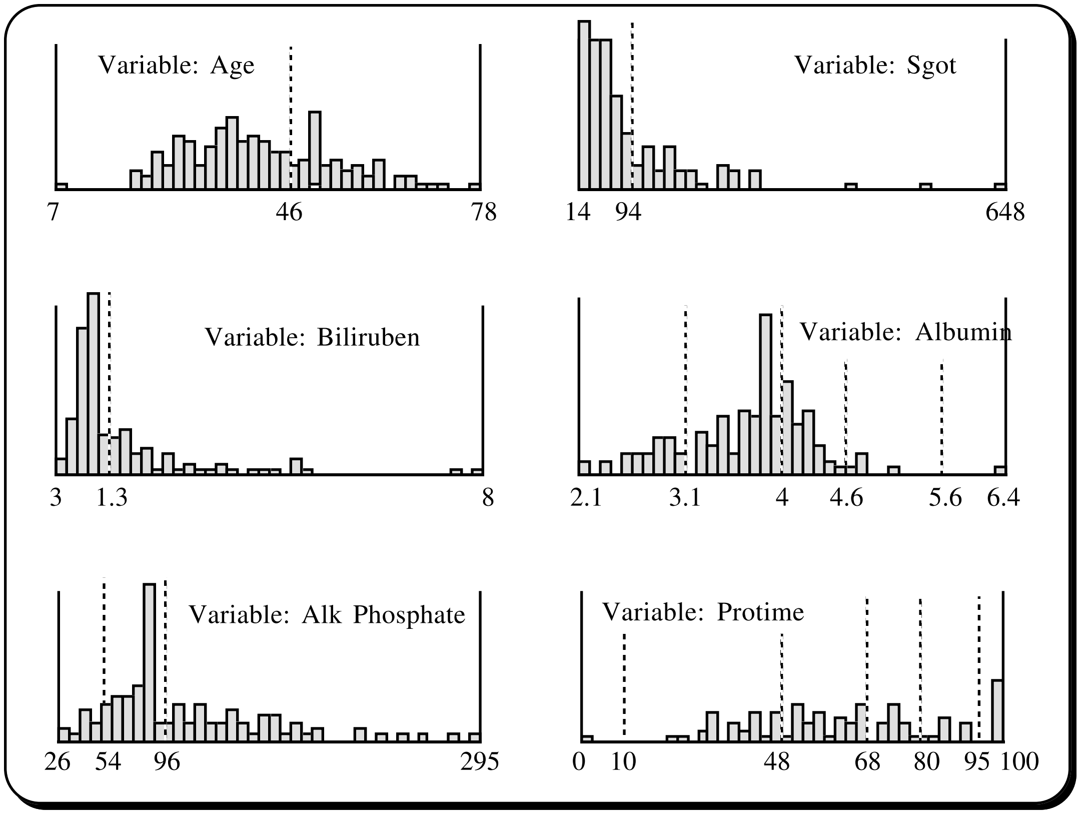

Figure 2.6. Continuously valued variables from hepatitis data set discretized by VALLEY

In a study of K-nearest neighbor clustering it was found that the K-NN algorithm performed well for discretizing real data--provided that the proper values for K and T (parameters corresponding to size of neighborhood and number of shared neighbors) were used. Unfortunately, there is no method for finding good K and T values other than experimentation. Further, these good values appear to be dependent on the initial data. Sample data from another UCI database on iris flowers was discretized using both K-NN and VALLEY. VALLEY found the same best clustering that K-NN did; however K-NN required repeated attempts with various values for K and T, via interaction with the user, as well as a manual evaluation of classification "goodness." VALLEY found the clustering in one iteration with no user intervention.

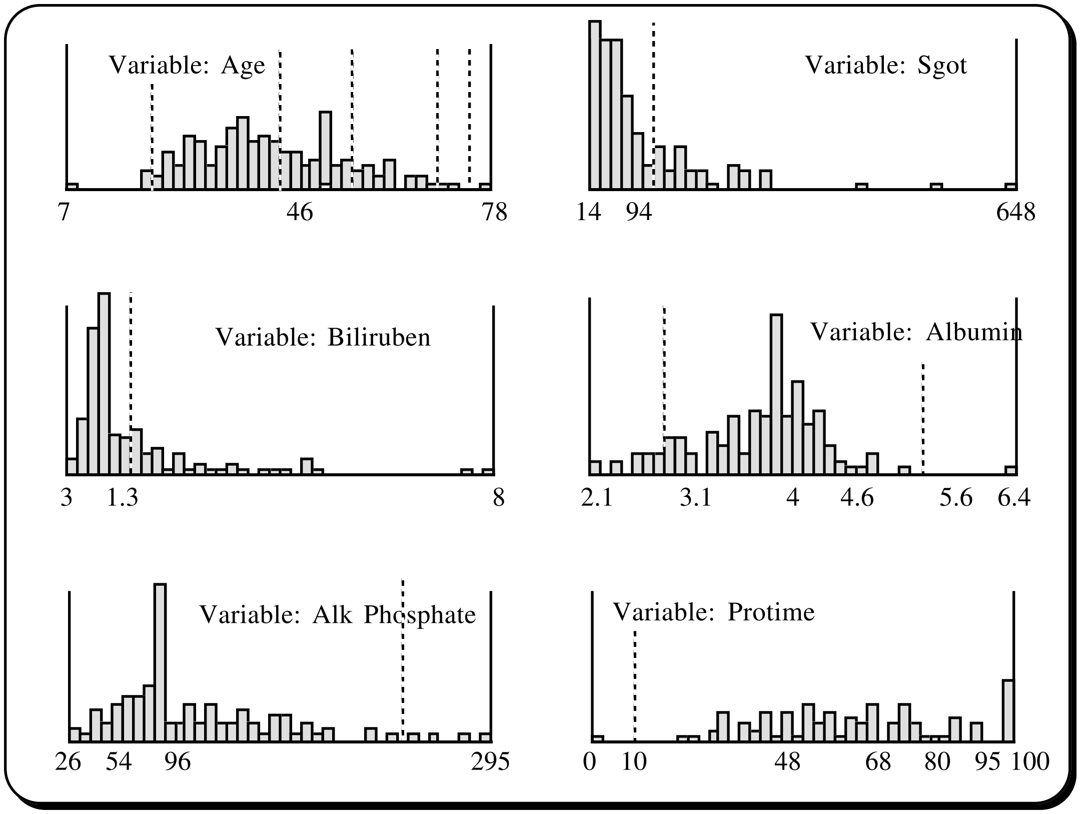

Figure 2.7. Continuously valued hepatitis data discretized by SLICE

The next natural step is to compare these methods with others by feeding the respective discretizations into a supervised learning system and noting effects on the system's ability to learn. Results from performing this type of comparison over 11 datasets from the UCI database are presented in the next chapter.

2.5. Analysis

Because BRACE is an abstract concept, meaningful analysis of it must be performed upon a concrete instantiation such as VALLEY or SLICE.

2.5.1. *Complexity Analysis.* Both algorithms have a time complexity of $O(nlogn)$, where n is the number of examples in the training set. The initial sort requires $nlogn$ steps, the ranking of boundaries requires n steps in VALLEY and no steps in SLICE, and the evaluation of different classifcations requires $|V|n$ and $|S|n$ steps respectively. But since $|V| \ll n$ and $|S| \ll n$, these simplify to n and we have $nlogn + n + n = O(nlogn)$ for VALLEY and $nlogn + n = O(nlogn)$ for SLICE.

2.5.2. *Pros and Cons.* BRACE and its two instantiations presented above meet most of the objectives of Table 1.2. The two notable exceptions are higher order correlations and feedback. This is not, however, a disadvantage when comparing BRACE with the other discretization methods discussed in this work because none of them include these capabilities. However, it is still believed that these capabilities are important and should be considered in discretization.

A further advantage is that of generality. BRACE may be applied to many different types of clustering and unsupervised learning tasks because of its dynamic Boundary Definition, Boundary Ranking Function, and Classification Optimization Function. Performance of BRACE will depend on the proper choice of BD, BRF, and COF for the data in question. Obviously, this is not a trivial task and many more instantiations of the BRACE paradigm may be evaluated as discretizing systems. The eventual automation of this aspect of the discretization process is a topic for future research.

Chapter 3
A Comparison of Six Discretization Methods

This chapter presents and discusses empirical results obtained by using six different discretization methods as preprocessors to three different supervised learners. The discretization methods used are ChiMerge, Equal Frequency Intervals, Maximin into Kmeans, Slice, Valley, and Equal Width Intervals. The supervised learners used are ID3 [Qui90], CN2 [Cla89], and Bounded Order Critical Feature Sets [And93].

3.1. Description of Supervised Learners Used

The three supervised learners used were chosen because they all require a discretizing preprocessor and represent different approaches to supervised learning. Two of the methods (ID3 and CN2) are fairly well-known and are successful machine learning approaches and the third is a technique being developed at Brigham Young University.

3.1.1. *ID3*. ID3 is a decision tree approach to learning. A decision tree is built using information gain as the metric for splitting nodes. This is a greedy approach that attempts to maximize information gain at each node by splitting on the attribute that currently provides the most information gain. The final hypothesis is represented by a decision tree. C4.5 [Qui93] is an extension of ID3 that, among its improvements, allows handling of continuously valued data by basically testing all possible splits over the range of the attribute to be discretized. This discretization is performed independently at each node in the tree. Therefore, discretizing the data before giving it to ID3 is similar to giving the undiscretized data to C4.5. This is convenient in the sense that we can compare results obtained by discretization and ID3 directly with C4.5 which performs its own internal handling of continuously valued data (see Table 3.6 and accompanying discussion).

3.1.2. *CN2*. CN2 is a rule induction algorithm that incorporates ideas from both ID3 and from AQ* [Mic90][Mic83]. Thus CN2 attempts an eclectic combination of the efficiency and ability to cope with noisy data that ID3 exhibits, with the flexible search strategy and if-then rule format of AQ*. The algorithm attempts to build good if-then rules by considering different combinations of variables and assessing how well each rule properly classifies the training instances. The search space is exponential and, therefore various heuristics are used to limit search time and space. The hypothesis is represented as an ordered list of if-then rules.

3.1.3. *Bounded Order Critical Feature Sets.* This is an experimental approach that assumes that most interesting attribute correlations (features) are of a relatively low order. Thus, even though examining all higher order correlations is an exponentially expensive proposition in terms of both time and space, according to the assumption most *interesting* correlations or features may be examined by brute force in bounded time. Therefore, the algorithm first considers all first order features and evaluates their ability to correctly classify the training set. It then considers all second order features and so on up to some relatively small *kth* order. Some subset of the features are then chosen to represent the hypothesis.

3.2. Empirical Results

Tables 3.1-3.3 summarize the results obtained by using each of the discretization methods as a preprocessor for each of the supervised learners. The data sets used were all obtained from the UC Irvine Machine Learning Database [Mur92], and each data set consists entirely of continuously valued inputs and has a single nominal output variable. A brief summary of each follows.

Glass. 9 inputs, 7 output classes. This is a collection of data from crime lab reports. The nine inputs are measures of mineral content (Mg, Al, etc.) and refractive index of different samples of glass. The problem is to determine what the glass was used for--windows, container, headlamp, etc.

Wine. 13 inputs, 3 output classes. These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivators. Analysis determined the quantities of 13 constituents found in each of the three types of wines and these constitute the 13 input variables. The problem is to determine which of the three cultivators produced the wine.

Vowel. 10 inputs, 11 output classes. Speech signals from 15 different speakers were low pass filtered at 4.7kHz and then digitized to 12 bits with a 10kHz sampling rate. Twelfth order linear predictive analysis was carried out on six 512 sample Hamming windowed segments from the steady part of the vowel. The reflection coefficients were used to calculate 10 log area parameters, giving a 10 dimensional input space. Based on the ten inputs, the problem is to determine which vowel sound was spoken.

Sonar. 60 inputs, 2 output classes. Each of the sixty inputs represents a measure of the energy within a particular frequency band. Given these inputs, determine whether the sonar image is a rock or a mine.

Iris. 4 inputs, 3 output classes. Given sepal length and width and petal length and width, determine which type of iris the flower is.

Bupa. 6 inputs, 2 output classes. The six inputs represent the outcome of five blood test results and number of drinks (half-pint equivalents of alcoholic beverages consumed per day) for a group of male Bupa Indians. The problem, given the input data, is to determine which of two "selector" classes the individual is in, where the selector classes attempt to divide the group as to likelihood of contracting a liver disorder.

Pima. 8 inputs, 2 output classes. The inputs include age, number of times pregnant, and the results of various medical tests and are taken from a population of female Pima Indians. Given these, determine if the individual tests positive for diabetes.

Vehicle. 18 inputs, 4 output classes. The inputs represesent silhouette features extracted by the HIPS (Hierarchical Image Processing System) extension BINATTS at the Turing Institute. Given these inputs the problem is to determine which of four vehicles the silhouette represents.

Results were obtained over ten runs by using 70% of the data from these data sets as a training set for learning and the remaining 30% as a test set. The results reported in the tables are percent accuracy on the test set.

	Glass	Wine	Vowel	Sonar	Iris	Bupa	Pima	Vehicle	Avg.
Chi	61	92	61	74	95	62	73	67	73
Freq	53	88	62	65	93	52	67	62	68
Kmmm	64	89	62	68	93	58	73	61	71
Slice	59	93	36	78	95	61	73	62	70
Valley	65	81	47	65	96	60	74	66	69
Width	58	92	64	77	95	51	70	62	71
Rankings:									
Chi	3	2	4	3	2	1	2	1	2.3
Freq	6	5	2	5	4	5	6	3	4.5
Kmmm	2	4	2	4	4	4	2	6	3.5
Slice	4	1	6	1	2	2	2	3	2.6
Valley	1	6	5	5	1	3	1	2	3.0
Width	5	2	1	2	2	6	5	3	3.3

Table 3.1. Performance and ranking of six discretization methods as a preprocessor to ID3

In each of the tables the data sets are represented in columns and the discretization methods are represented in rows. The tables are split into two sections. The top matrix of each gives the percent accuracy of the supervised learner on a given data set using a particular discretization method. For example, Table 3.1, which gives results for ID3, shows that ID3 achieved an accuracy of 61% on the vowel data set when ChiMerge was used as the preprocessor. Table 3.2 shows an accuracy of 48% for CN2 for the same data and discretization method and Table 3.3 show an accuracy of 81% for Bounded Order Critical Feature Sets.

The bottom matrix of each table shows the relative rank of each discretization method for each data set. Continuing the example above, Table 3.1 shows that for ID3's learning of the vowel data set, ChiMerge was the 4th best discretization method. Table 3.2 shows that for CN2, ChiMerge was again the 4th best choice, but Table 3.3 shows that for Bounded Order Critical Feature Sets, ChiMerge was the best discretization method of the six.

	Glass	Wine	Vowel	Sonar	Iris	Bupa	Pima	Vehicle	Avg.
Chi	64	75	48	78	95	63	71	45	67
Freq	51	83	57	79	64	57	69	53	64
Kmmm	64	88	57	78	93	56	72	59	71
Slice	59	72	33	80	96	62	74	55	66
Valley	53	75	35	74	92	61	72	43	63
Width	49	85	58	75	92	54	71	57	68
Rankings:									
Chi	1	4	4	3	2	1	4	5	3.0
Freq	5	3	2	2	6	4	6	4	4.0
Kmmm	1	1	2	3	3	5	2	1	2.3
Slice	3	6	6	1	1	2	1	3	2.9
Valley	4	4	5	6	4	3	2	6	4.0
Width	6	2	1	5	4	6	4	2	3.8

Table 3.2. Performance and ranking of six discretization methods as a preprocessor to CN2

Examining the data in all three tables yields some interesting results. First, a perusal of the rankings for each discretization method shows no clearly best method. Although overall ChiMerge appears to be a little more robust than the others, the best method of discretization appears to be data dependent. Also, the best discretization method may depend on the supervised learner for which the preprocessing is being performed. To see this, notice that ChiMerge does quite well for Bounded Order Critical Feature Sets but not nearly so well for CN2. K-means combined with Maxi-min (Kmmm) seems to perhaps be the most robust choice in this case.

	Glass	Wine	Vowel	Sonar	Iris	Bupa	Pima	Vehicle	Avg.
Chi	62	71	81	90	94	68	72	72	76
Freq	46	69	4	78	93	63	64	65	60
Kmmm	48	68	3	73	89	58	69	60	59
Slice	52	71	23	86	95	49	72	72	65
Valley	57	67	67	85	94	59	68	63	70
Width	39	65	8	78	94	53	70	65	59
Ranking:									
chi	1	1	1	1	2	1	1	1	1.1
freq	5	3	5	4	5	2	6	3	4.1
kmmm	4	4	6	6	6	4	4	6	5.0
slice	3	1	3	2	1	6	1	1	2.3
valley	2	5	2	3	2	3	5	5	3.4
width	6	6	4	4	2	5	3	3	4.1

Table 3.3. Performance and ranking of six discretization methods as a preprocessor to Bounded Order Critical Feature Sets

Table 3.4 summarizes the best, worst, and average accuracies obtained on the data sets by nine learning algorithms currently under research. For details on the algorithms and methods of obtaining these results see chapter 4 section 4 (p.53) and [Zar94]. A second result of interest is obtained by comparing accuracy results from Tables 3.1-3.3 with the results in Table 3.4. Although each supervised learner performs well on some of the datasets (at least assuming discretization done by the best method), each one also performs quite poorly on others even using the discretization method with the best results.

Dataset	Best	Average	Worst
Glass	68.6	66.4	61.6
Wine	93.8	92.6	90.5
Vowel	92.6	81.8	73.3
Sonar	78.9	75.7	72.5
Iris	95.3	94.7	94.0
Bupa	67.0	62.8	56.5
Pima	74.1	71.1	68.3
Vehicle	77.6	70.1	59.2

Table 3.4. Best, worst, and average performance for nine learning algorithms over the data sets [Zar94]

Table 3.5 supports this claim by comparing the three supervised learners' best results with the average and best results of the nine algorithms just mentioned. AvgC is the average percent correct for all nine algorithms and MaxC is the maximum percent correct for any of them (these values are taken directly from table 3.4). Bold table entries indicate relatively poor performance -- ID3 does comparatively poorly on the glass, vowel, bupa, and vehicle data sets; CN2 does comparatively poorly on the glass, wine, vowel, and vehicle datasets; and Bounded Order Critical Feature Sets performs comparatively poorly on the glass and wine data sets.

	Glass	Wine	Vowel	Sonar	Iris	Bupa	Pima	Vehicle
ID3	**65**	93	**64**	78	96	**62**	74	**67**
CN2	**64**	**88**	**58**	80	96	63	74	**59**
BOCFS	**62**	**71**	81	90	95	68	72	72
AvgC	66	93	82	76	95	63	71	70
MaxC	69	94	93	79	95	67	74	78

Table 3.5. Preprocessing vs. direct handling of continuously valued data

The claim that discretization as preprocessing is usually detrimental to performance is further substantiated by comparing the performance results of ID3 and the best preprocessor for each data set with C4.5 (Table 3.6). As may be seen from the table, for all data sets except sonar, discretization as a preprocessing step did not significantly improve performance. Again, for the same four data sets -- glass, vowel, iris, and vehicle -- the discretization was actually detrimental.

Dataset	ID3	C4.5
Glass	65	68
Wine	93	93
Vowel	64	80
Sonar	78	75
Iris	96	95
Bupa	62	65
Pima	74	73
Vehicle	67	70

Table 3.6 ID3 with (best) preprocessing vs. C4.5

It should be noted that the three algorithms examined here -- ID3, CN2, and BOCFS -- were designed to explore the idea of concept learning. Concept learning in general does not address the issue of continuously valued data and often deals exclusively with nominal data. Therefore, the lower performance results for these algorithms on data sets comprised entirely of continuously valued attributes is not surprising. This observation further strengthens the position that consideration of continously valued data should not be ignored in design of a learning algorithm.

These two results together, namely that 1) the best discretization method is dependent both on the application as well as on the supervised learner and that 2) in general, discretization as a preprocessing step can lead to severe performance degradation, provide evidence that discretization as a preprocessing step independent of the supervised learning system is not the appropriate approach to machine learning.

3.3. Problems with Discretization as a Preprocessing Step

There are at least two inherent problems in using discretization as a preprocessing step for a supervised learner. One of these problems results in the best discretization method being dependent on the application and supervised learner used. The other results in the sometimes poor overall performance observed.

3.3.1. *The Independence Problem.* This has to do with the fact that a general method of discretization has no idea what sort of information the supervised learning method requires to do its learning. Therefore, the discretizer may unwittingly eliminate some valuable (to the particular supervised learner in question) information in the preprocessing step thereby hampering its ability to learn. On the other hand, the same discretization for another supervised learner may have essentially avoided eliminating important information (at least this time).

3.3.2. *The Higher Order Correlation Problem.* Obviously, there are often higher order correlations between input variables. That is, one input variable may not tell us anything interesting about the output class by itself, but in combination with one or more of the other input variables, it may tell us a lot. The idea behind discretizing the data as a preprocessing step has been to leave the discovery of all these higher order correlations to the supervised learner. Unfortunately, in considering and discretizing each of the input variables independently, the discretizer may destroy the higher order correlation. For example, consider iris flowers. Perhaps sepal lengths fall into two general categories: long (those over 2 inches) and short (those 2 inches or less); however, perhaps if the sepal width is greater than .2 inches, then a sepal length is not considered long unless it is over 3 inches. Since the discretizer is not allowed to discover this higher order correlation, it will place the input variable sepal length into the 'long' interval if it is over 2 inches, whether that instance's sepal width is greater than .2 or not. It will therefore incorrectly discretize the sepal length variable of any instance whose sepal length is between 2 and 3 inches long and whose sepal width is greater than .2 inches.

The empirical results of this chapter as well as the two significant problems discussed above provide evidence that discretization as a separate preprocessing step can lead to severe performance degradation. The remaining chapters therefore adopt a different approach, abandoning to some degree, the idea that discretization can be handled completely independently of the supervised learning process.

Chapter 4
Statistical Prototypes

The results of chapter 3 indicate a need for a different approach to discretization. First, discretization must consider higher-order correlations between input variables. Second, the discretization process must not, due to its independence, destroy information valuable to the supervised learner. This chapter presents an attempt to rectify both of these problems while still maintaining the autonomy (in many cases) of the discretization process. The basic approach is to use a standard minimum distance classification scheme with a single prototype per class [Tou74], which works well for simple problems. However, this must be extended to multiple prototypes per class in the general case. The MSP algorithm is a unique, nonparametric approach to creating multiple prototypes per class.

4.1. Introduction

The idea of using prototypes to represent classes is not a new one [Aha91][Aha89][Das91][Rum86][Tou74]. It is a simple and natural approach to the problem of dealing with continuously valued attributes. The basic assumption is that given an m-dimensional space defined by the input variables, there exists one or more representative points in that space for each output class. These representative points are termed prototypes. Statistical prototypes (SP) are a simple variation on this idea. They assume that all input variables are continuously valued and that each output class can be represented by one or more gaussian functions over these input variables. This assumption is not unreasonable because all functions may be approximated by one or more gaussian bases, the worst case being the degenerate one in which each data point is represented by it's own gaussian function with mean equal to the value of the data point and standard deviation zero. The idea of using statistical information obtained from a training set in the formation of prototypes is not a novel idea, either. Two examples of similar systems are radial basis function networks [Hay94] and CLASSIT [Gen90].

CLASSIT is an unsupervised concept learning system based on the same model as Fischer's COBWEB [Fis90][Gen90]. Both CLASSIT and COBWEB create a hierarchy of concept descriptions in an attempt to discover natural classes in the training data. Basic learning begins with the presentation of an instance to the system, which then chooses from four possible operations: incorporate the instance into a given concept, create a new concept, split an existing concept, or merge two existing concepts. All four possibilities are tried with their respective successes being measured by a concept utility function:

$$\frac{\sum_{k}^{K} P(C_k) \sum_{i}^{I} 1/\sigma_{ik} - \sum_{i}^{I} 1/\sigma_{ip}}{K}$$

where I is the number of attributes, K is the number of classes (concepts) in the partition, σ_{ik} is the standard deviation for a given attribute in a given class and σ_{ip} is the standard deviation for a given attribute in the parent node. The operation that maximizes the utility is incorporated and the process is repeated for the next instance. Two parameters, *acuity* and *cutoff*, control the breadth and depth, respectively, of the concept hierarchy. The concepts are described by an attribute vector that represents the mean values for instances that belong to the concept.

Radial basis function (RBF) networks are a connectionist approach to learning concepts that may employ supervised or unsupervised learning techniques or a hybrid of the two. RBF networks are multi-layer nets in which the hidden layer of nodes represent a set of radial basis functions that perform a nonlinear mapping of the inputs to a higher dimensional function space and the output units then perform a linear mapping from that space to the output space. This nonlinear mapping to higher dimensionality is an attempt to "force" natural clusterings in the data to appear that are not readily apparent in spaces of lower dimensionality. While these functions may be any type of radial basis function, in practice they quite often are a simple multi-variate gaussian distribution. Training of the network includes two parts: the learning of the correct distributions in the hidden layer that will perform the nonlinear mapping on the input, and the training of the output units to perform a linear mapping (similar to a simple perceptron) on the result.

The statistical prototypes (SP) presented here differ from CLASSIT in their supervised approach to learning, their utility measure (distance metric), and in the fact that SP does not use a merge-type operation. SP and RBF networks have significantly different representations for hypotheses, SP learning is generally faster, and SP does not employ the idea of using a non-linear mapping into a higher dimensionality.

4.2. Creating Statistical Prototypes

To begin with, each output class will be assumed to be represented, roughly, with a single m-dimensional normal distribution over the input space. Therefore, by assumption, each output class will be representable with a single prototype.

Define:

T as a set of training instances;
n as the number of instances in T, that is $n = |T|$;
c as the number of output classes represented in T;
v as the number of input variables represented in T;
i as an index that ranges $0 \le i < c$ and indicates output class;
j as an index that ranges $0 \le j < v$ and indicates input variable;
T_i as the c subtraining sets obtained by partitioning T by output class;
o_i as the c output classes;
m_{ij} as the mean of the *jth* input variable for T_i;
σ_{ij}, as the standard deviation of the *jth* input variable for T_i;
$\boldsymbol{p}_i$ as the prototype for class i;
$\boldsymbol{x}$ as a vector of inputs representing an instance;
x_j as the *jth* input of an instance;
d_i as the normal distance between a point $\boldsymbol{x}$ and $\boldsymbol{p}_i$.

The basic algorithm (Algorithm 4.1) is outlined below and proceeds as follows. First, divide the training set T by output class, creating c subtraining sets T_i. For each T_i and each input variable j calculate m_{ij} and σ_{ij}. $\boldsymbol{p}_i$ is a vector of ordered pairs (m_{ij}, σ_{ij}). Classification of a new instance $\boldsymbol{x}$ is accomplished by calculating all d_i for the point defined by that instance and outputting z such that $d_z = min\{d_i\}$, where d_i is calculated as

$$d_i = \sum_{j=0}^{v} \frac{|x_j - m_{ij}|}{\sigma_{ij}}.$$

Two points are worth a brief mention here. First, this is obviously a variation of a standard minimum distance classifier [Tou74]. Thus, SP is not a novel idea; however, it is the basis for MSP, which extends the minimum distance classifier to multiple prototypes per class in a novel way, and is thus important as background material. Second, the name Statistical Prototypes may be somewhat misleading. SP is not a statistical approach to clustering/classification in the Bayesian sense of relying on the statistical properties of the underlying distributions [Tou74]. It is statistical only in the sense of employing the simple statistical variables of mean and standard deviation in representing prototypes.

Learn()
  Create subtraining sets T_i
  Calculate m_{ij} and σ_{ij}
  Create $\boldsymbol{p}_i$ as j ordered pairs, (m_{ij}, σ_{ij})

Classify($\boldsymbol{x}$)
  calculate all d_i
  output z such that $d_z = min\{d_i\}$

Algorithm 4.1. Single prototype per output class (SP)

4.2.1. *An Example.* Suppose that T consists of the following set of nine instances:

Petal Wid.	Petal Len.	Sepal Wid.	Sepal Len.		
6.0	3.0	4.8	1.8	->	virginica,
6.3	2.5	5.0	1.9	->	virginica,
4.9	2.5	4.5	1.7	->	virginica,
4.6	3.1	1.5	0.2	->	setosa,
4.8	3.1	1.6	0.2	->	setosa,
5.3	3.7	1.5	0.2	->	setosa,
5.1	2.5	3.0	1.1	->	versicolor,
5.2	2.7	3.9	1.4	->	versicolor,
5.9	3.0	4.2	1.5	->	versicolor.

Then $c = 3$, $v = 4$, and o_0 is virginica, o_1 is setosa, and o_2 is versicolor. T_0 consists of the first three instances, T_1 of the second three instances, and T_2 of the last three instances. Calculating the means and standard deviations we get

$m_{00} = 5.733$, $\sigma_{00} = .602$,
$m_{01} = 2.667$, $\sigma_{01} = .236$,
$m_{02} = 4.766$, $\sigma_{02} = .205$,
$m_{03} = 1.800$, $\sigma_{03} = .082$,
$m_{10} = 4.900$, $\sigma_{10} = .294$,
$m_{11} = 3.300$, $\sigma_{11} = .283$,
$m_{12} = 1.533$, $\sigma_{12} = .047$,
$m_{13} = 0.200$, $\sigma_{13} = .001$*,
$m_{20} = 5.400$, $\sigma_{20} = .356$,
$m_{21} = 2.733$, $\sigma_{21} = .205$,
$m_{22} = 3.700$, $\sigma_{22} = .510$,
$m_{23} = 1.333$, $\sigma_{23} = .189$.

*Note that σ_{13} is really equal to 0. However, to avoid a division by 0 in the calculation of d_i, any standard deviation that equals 0 is set to an arbitrarily small substitute.

The notation for expressing a prototype used throughout will be

$$\boldsymbol{p}_i = \begin{array}{cccc} m_{i0} & m_{i1} & \ldots & m_{iv-1} \\ \sigma_{i0} & \sigma_{i1} & \ldots & \sigma_{iv-1} \end{array} \quad \text{->}\ o_i.$$

Therefore,

$$\boldsymbol{p}_0 = \begin{array}{cccc} 5.733 & 2.667 & 4.766 & 1.800 \\ 0.602 & 0.236 & 0.205 & 0.082 \end{array} \quad \text{-> virginica,}$$

$$\boldsymbol{p}_1 = \begin{array}{cccc} 4.900 & 3.300 & 1.533 & 0.200 \\ 0.294 & 0.283 & 0.047 & 0.001 \end{array} \quad \text{-> setosa,}$$

$$\boldsymbol{p}_2 = \begin{array}{cccc} 5.400 & 2.733 & 3.700 & 1.333 \\ 0.356 & 0.205 & 0.510 & 0.189 \end{array} \quad \text{-> versicolor.}$$

Now, suppose an instance $\boldsymbol{x} = (5.0, 2.9, 2.9, 1.1)$ is presented to the system for classification. Distances from $\boldsymbol{x}$ to each prototype are calculated:

$$d_0 = \frac{|5 - 5.733|}{.602} + \frac{|2.9 - 2.667|}{.236} + \frac{|2.9 - 4.766|}{.205} + \frac{|1.1 - 1.8|}{.082} = 19.84,$$
$$d_1 = \frac{|5 - 4.900|}{.294} + \frac{|2.9 - 3.3|}{.283} + \frac{|2.9 - 1.533|}{.047} + \frac{|1.1 - .2|}{.001} = 930.84,$$
$$d_2 = \frac{|5 - 5.4|}{.356} + \frac{|2.9 - 2.733|}{.205} + \frac{|2.9 - 3.7|}{.510} + \frac{|1.1 - 1.33|}{.189} = 4.74.$$

The prototype for o_2 or versicolor is the closest to the example so $\boldsymbol{x}$ is classified as versicolor.

4.2.2. *Empirical Results.* The data sets described previously were processed using statistical prototypes and the results are shown in Table 4.2. These results represent an average taken over 25 runs. The first column shows the percentage of test instances classified correctly using statistical prototypes. As before, 70% of the data were used as training instances, and the remaining 30% as test instances. The remaining three columns reproduce the results of Table 3.4 as a reference.

Dataset	SP	Best	Average	Worst
Glass	**49.3**	68.6	66.4	61.6
Wine	**94.6**	93.8	92.6	90.5
Vowel	**46.8**	92.6	81.8	73.3
Sonar	**73.3**	78.9	75.7	72.5
Iris	**95.0**	95.3	94.7	94.0
Bupa	**59.9**	67.0	62.8	56.5
Pima	**69.1**	74.1	71.1	68.3
Vehicle	**47.5**	77.6	70.1	59.2
Thyroid	**94.5**	93.5	92.1	89.2
Waveform	**79.5**	72.0	68.7	66.0
Wavenoise	**81.6**	73.0	68.8	65.0

Table 4.2. Using single statistical prototypes to classify the data sets

Notice that statistical prototypes do very well in comparison with other learning models on the iris, thyroid, waveform, wavenoise, and wine data sets. Since a single prototype represents each class this indicates that these are fairly simple data sets to learn and represent essentially linearly separable problems. Statistical prototypes perform fairly well on the bupa, pima, and sonar data sets. This indicates that these problems, too, are fairly easy although not quite linearly separable, since other, more advanced, methods perform significantly better on them. Creating multiple prototypes per class could improve performance. Finally, statistical prototypes do quite poorly on the glass, segment, vehicle, and vowel data sets. It appears that these data sets are far from linearly separable and thus a single prototype per class representation would be expected to perform poorly. Multiple prototypes per class are a necessity in these cases.

4.3. Extension to Multiple Prototypes Per Class (MSP)

In many cases a single prototype will not be sufficient for representing an entire output class. Extending single-prototype minimum distance classifiers to multiple prototypes is a difficult problem because there exists no accepted basis for determining how many prototypes to create. Various approaches to the problem include instance based learning [Aha89][Aha91], nearest neighbor methods [Das91], and clustering techniques such as ISODATA [Tou74]. These techniques are mainly differentiated by their heuristics for determining how to create a prototype and when the proper number of prototypes have been created. Multiple statistical prototypes (MSP) employ an error feedback heuristic to extend the simple statistical prototype described in the previous section. The extension requires two modifications. First, add the new definitions:

ε_b as the error in classifying the training set before creating a new prototype;
ε_a as the error in classifying the training set after temporarily splitting a prototype;
ε_{small} as the smallest error in classifying the training set after temporarily splitting each prototype;
q_i as the number of prototypes for output class i;
k as an index that ranges $0 \le k < q_i$ and indicates prototype number;
i_{small}, j_{small}, k_{small} as the values of i, j, and k during temporary splitting of prototypes that result in the value ε_{small};
q_{small} as $q_{i_{small}}$, a slight abuse of notation;
T_i^k as the sub-training set that contains all the instances of output class i that are closest to the kth prototype for class i;
t_i^k as an instance in T_i^k;
$min\{j, T_i^k\}$ as the minimum value of input j for any t_i^k in T_i^k;
$max\{j, T_i^k\}$ as the maximum value of input j for any t_i^k in T_i^k.

Also, replace the definitions for $\boldsymbol{p}_i$ and d_i by the following:

$\boldsymbol{p}_i^k$ as the *kth* prototype for class i;

d_i^k as the normal distance between a point and $\boldsymbol{p}_i^k$.

Second, a method for determining how many prototypes are required for an output class is required. This may be approached in many different ways. In this case, a simple greedy algorithm, algorithm 4.3, is applied as follows.

First, run algorithm 4.1 to generate a single prototype, $\boldsymbol{p}_i^0$, for each output class. Next, calculate ε_b by using the prototypes to attempt to classify the training set T. Then temporarily split $\boldsymbol{p}_0^0$ on input 0 as follows. Find $min\{0, T_0^0\}$ and $max\{0, T_0^0\}$ and temporarily divide T_0^0 into T_0^{0min} and T_0^{0max} where T_0^{0min} contains all t_0^0 closer to $min\{0, T_0^0\}$ than to $max\{0, T_0^0\}$, and T_0^{0max} contains all t_0^0 closer to $max\{0, T_0^0\}$ than to $min\{0, T_0^0\}$. Ties may be decided arbitrarily. Calculate temporary prototypes $\boldsymbol{p}_0^{0min}$ and $\boldsymbol{p}_0^{0max}$ as before and then calculate ε_a by again attempting to classify T, the only change being that $\boldsymbol{p}_0^0$ has been temporarily replaced by $\boldsymbol{p}_0^{0min}$ and $\boldsymbol{p}_0^{0max}$. Reunite $\boldsymbol{p}_0^{0min}$ and $\boldsymbol{p}_0^{0max}$ into $\boldsymbol{p}_0^0$ and repeat for all inputs x_j and all prototypes $\boldsymbol{p}_i^k$, in order to find the values i_{small}, j_{small}, k_{small}, and ε_{small}. Now, if $\varepsilon_{small} < \varepsilon_b$, permanently split $\boldsymbol{p}_{i_{small}}^{k_{small}}$ into $\boldsymbol{p}_{i_{small}}^{k_{small}}$ and $\boldsymbol{p}_{i_{small}}^{q_{small}}$, increment q_{small}, and set $\varepsilon_b = \varepsilon_{small}$. Repeat the entire process until $\varepsilon_{small} \geq \varepsilon_b$. Classification of new instances is performed the same as in algorithm 4.1.

Learn()
    Run Algorithm 4.1
    Calculate ε_b
    While($\varepsilon_{small} < \varepsilon_b$)
        For $i = 0$ to c-1
            For $k = 0$ to q_i-1
                For $j = 0$ to v-1
                    Split($\boldsymbol{p}_i^k$,j)
                    Calculate ε_a
                    If $\varepsilon_a < \varepsilon_{small}$
                        $\varepsilon_{small} = \varepsilon_a$
                        $i_{small} = i$
                        $j_{small} = j$
                        $k_{small} = k$
                    Unsplit($\boldsymbol{p}_i^k$,j)
        If $\varepsilon_{small} < \varepsilon_b$
            Split($\boldsymbol{p}_{i_{small}}^{k_{small}}$,$j_{small}$)
            $q_{small} = q_{small}$+1
            $\varepsilon_b = \varepsilon_{small}$

Split($\boldsymbol{p}_i^k$,j)
    $T_i^{kmin} = \{ t_i^k \mid t_i^k$ is closer to $min\{j, T_i^k\}$ than to $max\{j, T_i^k\}\}$
    $T_i^{kmax} = \{ t_i^k \mid t_i^k$ is closer to $max\{j, T_i^k\}$ than to $min\{j, T_i^k\}\}$
    calculate $\boldsymbol{p}_i^{kmin}$ and $\boldsymbol{p}_i^{kmax}$
    replace $\boldsymbol{p}_i^k$ with $\boldsymbol{p}_i^{kmin}$ and $\boldsymbol{p}_i^{kmax}$

Unsplit($\boldsymbol{p}_i^{kmin}$, $\boldsymbol{p}_i^{kmax}$)
    $T_i^k = T_i^{kmin} \cup T_i^{kmax}$
    calculate $\boldsymbol{p}_i^k$
    replace $\boldsymbol{p}_i^{kmin}$ and $\boldsymbol{p}_i^{kmax}$ with $\boldsymbol{p}_i^k$

Classify($\boldsymbol{x}$)
    calculate all d_i
    output z such that $d_z = min\{d_i\}$

Algorithm 4.3. Multiple prototypes per output class (MSP)

4.3.1. *An Example*. Suppose that T consists of the following instances:

4.8	1.8	->	negative,
5.0	1.9	->	negative,
4.5	1.7	->	negative,
3.5	1.2	->	negative,
3.6	1.2	->	negative,
3.5	1.2	->	positive,
3.0	1.1	->	positive,
3.9	1.4	->	positive,
4.2	1.5	->	positive.

Then initial calculation of prototypes will yield

$$\boldsymbol{p}_0^0 = \begin{matrix} 4.280 & 1.560 & & \\ 0.618 & 0.301 & \text{->} & \text{negative,} \end{matrix}$$

$$\boldsymbol{p}_1^0 = \begin{matrix} 3.650 & 1.300 & & \\ 0.450 & 0.158 & \text{->} & \text{positive.} \end{matrix}$$

The distances from all prototypes for each instance are presented in the following table:

	d_0	d_1
(4.8, 1.8)	1.639	5.720
(5.0, 1.9)	2.295	6.797
(4.5, 1.7)	0.821	4.421
(3.5, 1.2)	2.459	0.966
(3.6, 1.2)	2.296	0.744
(3.5, 1.2)	2.459	0.966
(3.0, 1.1)	3.599	2.710
(3.9, 1.4)	1.146	1.411
(4.2, 1.5)	0.329	2.488.

Now, T is classified using these prototypes and the results are shown below with the correct classification in parentheses.

4.8	1.8	->	negative (negative),
5.0	1.9	->	negative (negative),
4.5	1.7	->	negative (negative),
3.5	1.2	->	positive (negative),
3.6	1.2	->	positive (negative),
3.5	1.2	->	positive (positive),
3.0	1.1	->	positive (positive),
3.9	1.4	->	negative (positive),
4.2	1.5	->	negative (positive).

Thus, with a single prototype per class, ε_b = 4/9 = .444. Now, $\boldsymbol{p}_0^0$ is temporarily split on input 0 into $\boldsymbol{p}_0^{0min}$ and $\boldsymbol{p}_0^{0max}$. $Min\{x_0 \text{ of } t_0^0\}$ = 3.5 and $max\{x_0 \text{ of } t_0^0\}$ = 5.0. Therefore, dividing T_0^0 results in T_0^{0min} containing

4.8	1.8	->	negative,
5.0	1.9	->	negative,
4.5	1.7	->	negative,

and T_0^{0max} containing

3.5	1.2	->	negative,
3.6	1.2	->	negative.

Temporarily recalculating prototypes gives

$\boldsymbol{p}_0^{0min}$=	4.767	1.800		
	0.205	0.082	->	negative,
$\boldsymbol{p}_0^{0max}$=	3.550	1.200		
	0.055	0.001	->	negative,
$\boldsymbol{p}_1^0$ =	3.650	1.300		
	0.450	0.158	->	positive.

The new table of distances now includes distances from all three prototypes for each instance:

	d_{0min}	d_{0max}	d_1
(4.8, 1.8)	0.161	622.73	5.720
(5.0, 1.9)	2.356	726.36	6.797
(4.5, 1.7)	2.522	517.27	4.421
(3.5, 1.2)	13.496	0.909	0.966
(3.6, 1.2)	13.001	0.909	0.744
(3.5, 1.2)	13.496	0.909	0.966
(3.0, 1.1)	17.156	110.00	2.710
(3.9, 1.4)	9.107	206.36	1.411
(4.2, 1.5)	6.424	311.82	2.488.

Now, T is again classified, this time with the modified prototypes:

4.8	1.8	->	negative (negative),
5.0	1.9	->	negative (negative),
4.5	1.7	->	negative (negative),
3.5	1.2	->	negative (negative),
3.6	1.2	->	positive (negative),
3.5	1.2	->	negative (positive),
3.0	1.1	->	positive (positive),
3.9	1.4	->	positive (positive),
4.2	1.5	->	positive (positive).

So ε_a = 2/9 = .222. Since this is the first temporary split, ε_{small} is set equal to .222. $\boldsymbol{p}_0^{0min}$ and $\boldsymbol{p}_0^{0max}$ are now reunited into $\boldsymbol{p}_0^0$, and whole process is repeated for $\boldsymbol{p}_0^0$ on input 1, $\boldsymbol{p}_1^0$ on input 0 and $\boldsymbol{p}_1^0$ on input 1. Since ε_{small} is already less than ε_b, at least one permanent split will occur. If none of the other temporary splits classify better than 77.8% (7/9) correctly, $\boldsymbol{p}_0^0$ will be split on input 0.

4.3.2. *Empirical Results.* The data sets were reclassified using algorithm 4.3 in order to generate multiple prototypes per class. These results are shown in Table 4.4, which is an expansion of Table 4.2 that includes the new results.

Dataset	SP	MSP	Best	Average	Worst
Glass	49.3	**65.5**	68.6	66.4	61.6
Wine	94.6	**96.2**	93.8	92.6	90.5
Vowel	46.8	**72.7**	92.6	81.8	73.3
Sonar	73.3	**79.4**	78.9	75.7	72.5
Iris	95.0	**95.9**	95.3	94.7	94.0
Bupa	59.9	**60.5**	67.0	62.8	56.5
Pima	69.1	**71.3**	74.1	71.1	68.3
Vehicle	47.5	**64.0**	77.6	70.1	59.2
Thyroid	94.5	**95.4**	93.5	92.1	89.2
Waveform	79.5	**79.8**	72.0	68.7	66.0
Wavenoise	81.6	**81.1**	73.0	68.8	65.0

Table 4.4. Using multiple statistical prototypes to classify the data sets

As is to be expected, little improvement in performance was achieved for the data sets on which single prototypes performed well. However, dramatic performance increases are evident on the data sets on which single prototypes performed poorly.

4.4. Analysis

MSP was compared empirically with the following nine well-known learning models:

ID3 - A set of instances is used to induce a decision tree where each node comprises a test on a single attribute and the branches out of the node represent the different outcomes of the test. In constructing the tree, attributes are chosen and nodes are created on the basis of maximizing a theoretical measure of information gain [Qui90].

Ctree - C4.5 is an extension of ID3 that allows handling of noisy data and continuously valued data [Qui93].

Crule - C4.5 has the ability to represent the hypothesis as an ordered rule list instead of a tree. Whereas Ctree uses the traditional tree-structured hypothesis, Crule uses the alternate rule-list-structure [Qui93].

Cart - Classification and Regression Trees is a very general method of tree-structured learning. It's most basic ideas deal with how to construct a tree representation of the data provided in a training set, with the key ingredient being how to partition the data set at each node in the tree. Various methodologies for doing so are sampled, including statistical and Bayesian approaches as well as techniques from regression theory [Bre93].

Bayes - This learning algorithm basically uses the standard Bayesian approach of minimizing risk. That is, there is a loss associated with every output class with respect to every other output class, L(t, z), which is the loss or penalty associated with misclassifying class t as class z. Each decision rule, d, in the system has associated with it a risk, R(d) = expected loss of rule d. Decision rules are always chosen so as to minimize the risk, R(d) [Bre93].

MML - Minimum Message (Description) Length, introduced by Jorma Rissanen, approaches learning as a minimization of the number of bits needed to write down the observed data. This written description of the data includes not only a description of the examples seen, but also a description of a statistical model that explains the data--the parameters used, their values and even how they are used in the model. MML attempts to find the simplest description (in terms of number of bits) that expresses the observed data [Ris83].

IB3 - David Aha has introduced several instance-based learning algorithms. This is his Instanced-Based Algorithm 3 (also referred to as NTGrowth). In particular, this approach to learning addresses the issue of noisy data. Only those instances from the training set that are incorrectly classified are stored in the concept description. In addition, a count of how well these stored instances will perform is kept and any stored instance that does sufficiently poorly is eventually dropped from the concept description [Aha91].

IB4 - Instance-Based Algorithm 4 (also referred to as Bloom) is another of Aha's algorithms. This extends IB3 to address the issue of irrelevant attributes. Each target concept has a weight vector associated with it. The weight vector warps the function space, giving larger weights and thus more influence to those attributes that have been found to be most important in the function to be learned [Aha89].

CN2 - This is a mixture of ideas from Quinlan's ID3 and Michalski's AQ* [Mic90][Mic83]. The algorithm attempts to build good if-then rules by considering different combinations of attributes and determining how well each rule classifies the training set. Various heuristics are employed to limit the search space to a tractable size. Rule sets may be either ordered or unordered and for this comparison, unordered rule sets were used [Cla89].

Statistical prototypes outperform all models on six of the data sets (iris, sonar, thyroid, waveform, wavenoise, and wine) and perform near or above average on another three (bupa, glass, and pima). The vehicle and vowel data sets, however, demonstrate that MSP is still somewhat fragile, as it performed significantly lower than average on these. Table 4.5 compares multiple statistical prototypes with the other nine learning models by giving each a relative ranking for each data set according to how well that model performed on the data set. For example, MSP is ranked 1 on the waveform data set since it performed better than any other learning model on this data set. On the other hand, MSP is ranked 4 on the pima data set since three other models performed better in this case.

	Ctree	Crules	Cart	ID3	Bayes	MML	IB3	IB4	CN2	**MSP**
Glass	3	4	6	2	1	4	9	10	8	**7**
Wine	5	2	10	5	5	3	5	4	9	**1**
Vowel	5	8	9	3	4	7	1	2	10	**9**
Sonar	5	6	6	4	2	3	9	8	10	**1**
Iris	4	4	8	8	4	4	2	2	10	**1**
Bupa	2	7	1	4	5	2	8	10	9	**6**
Pima	3	9	2	8	5	1	7	5	10	**4**
Vehicle	5	10	6	2	3	4	9	1	7	**8**
Thyroid	10	8	8	4	7	6	3	4	2	**1**
Waveform	6	9	10	8	7	5	3	2	4	**1**
Wavenoise	6	9	8	2	4	3	5	7	10	**1**
	4.9	7.1	6.7	4.5	4.3	3.8	5.5	5.0	8.1	**3.6**

Table 4.5. Relative ranking of the models

The bottom row of the table shows average relative ranking. Even though MSP performs relatively poorly on two of the data sets, it is still ranked the best overall.

Table 4.6 shows the average number of prototypes created by MSP for each data set. The first column indicates the number of output classes for the appropriate data set. This may be thought of as a lower bound on the number of prototypes required by MSP. The second column indicates the average number of prototypes actually created by MSP. The third column indicates the number of instances in the training set (an upper bound on the number of prototypes), and the fourth column is the ratio #Prototypes/#Instances. This ratio gives an indication of parsimony, indicating to what degree the training information was able to be assimilated into a concise representation. As may be seen from the table, a high degree of parsimony is achieved in all cases. For all data sets, MSP created no more than an average of three prototypes per output class, which always resulted in at least a 90% reduction in information stored.

Dataset	#Classes	#Prototypes	#Instances	%Ratio
Glass	**7**	**13.71**	**150**	**9.1%**
Wine	3	5.30	125	4.2%
Vowel	**11**	**30.44**	**370**	**8.2%**
Sonar	2	6.04	146	4.1%
Iris	3	4.56	105	4.3%
Bupa	**2**	**3.48**	**242**	**1.4%**
Pima	2	7.12	538	1.3%
Vehicle	**4**	**11.84**	**593**	**2.0%**
Thyroid	3	5.56	1960	0.3%
Waveform	3	6.80	1400	0.5%
Wavenoise	3	7.28	1400	0.5%

Figure 4.6. Number of prototypes created by MSP

The bold entries indicate data sets on which MSP performs relatively poorly. Encouragingly, further testing of the statistical prototype concept has resulted in much better learning potential for those data sets on which MSP performs poorly. For example on the vowel data set, manually forcing the creation of more prototypes (up around 40), results in performance above 90%. The vehicle and glass data sets showed similar improvement. This suggests that for these data sets MSP prematurely terminated prootype splitting and that a different metric, rather than the simple greedy one employed ($\varepsilon_{small} < \varepsilon_b$) may yield even better results. Interestingly, only the bupa data set appeared resistant to further performance improvements. This is probably attributable to the data being less amenable to approximation using gaussian functions.

4.4.1. *Complexity Analysis*. Time complexity for SP, which employs a single prototype per output class is reasonable. The calculation of all prototypes requires two passes over the data set. During the first pass, all m_{ij} are calculated. During the second pass, all σ_{ij} are calculated. Since there are n instances in the training set and each instance contains v input variables, the entire process requires $2nv$ steps which yields $O(nv)$.

MSP, which allows multiple prototypes per output class, initially requires the same $2nv$ steps as does SP since it actually first runs SP. Then each temporary split requires $O(n+nv)$ steps -- $O(n)$ steps to divide the instances between $\boldsymbol{p}_i^{kmin}$ and $\boldsymbol{p}_i^{kmax}$ and then $O(nv)$ to calculate the temporary means and standard deviations. During each pass through the "while" loop all prototypes are split in all dimensions. This means $O(nv)$ splits since the number of prototypes is bounded above by n, the number of instances. Therefore, a single pass through the while loop will require $O(nv(n+nv)) = O(n^2v+n^2v^2)) = O(n^2v^2)$. Finally, since each pass through the while loop yields a new prototype and since the number of prototypes is bounded above by n, no more than n passes through the loop will be made. Therefore, the entire algorithm is $O(n(n^2v^2)) = O(n^3v^2)$.

4.5. SP/MSP and Nominal Data

This work began with an attempt at handling continuously valued data. Statistical prototypes provide this ability; however, in their present forms, the algorithms for statistical prototypes do not employ any method for handling nominally valued data. Obviously, the statistical methods presented cannot have any relevance to data that has no inherent linear order. Since many data sets contain nominally valued input variables as well as continuously valued input variables, this aspect of learning can no more be ignored than can the continuously valued one.

In the case where a data set contains only continuously valued input variables, statistical prototypes have performed effectively. In the case where a data set contains nominally valued input variables, two options present themselves. First, statistical prototypes may still be employed, as originally intended, as a preprocessing step to another learning model, M. In this scenario, SP/MSP would handle the continuously valued inputs and the other model would handle the nominally valued ones.

4.5.1. *An Example*. Suppose a training set T consisted of the following instances:

6.0	blue	sometimes	1.8	->	toxic,
6.3	green	always	1.9	->	toxic,
4.9	green	always	1.7	->	toxic,
4.6	blue	never	0.2	->	nontoxic,
4.8	blue	sometimes	0.2	->	toxic,
5.3	red	never	0.2	->	nontoxic,
5.1	green	sometimes	1.1	->	nontoxic,
5.2	blue	sometimes	1.4	->	nontoxic,
5.9	green	always	1.5	->	nontoxic.

Since the second and third inputs are nominally valued (the third may be considered ordered), SP/MSP can only use the first and fourth input. Using these, SP/MSP creates its own modified training set T' that contains only continuously valued inputs:

6.0	1.8	->	toxic,
6.3	1.9	->	toxic,
4.9	1.7	->	toxic,
4.6	0.2	->	nontoxic,
4.8	0.2	->	toxic,
5.3	0.2	->	nontoxic,
5.1	1.1	->	nontoxic,
5.2	1.4	->	nontoxic,
5.9	1.5	->	nontoxic.

SP/MSP then runs normally and creates its prototypes. Assume that a single prototype for each class is created. Then o_0 is toxic and o_1 is nontoxic and

$$\boldsymbol{p}_0 = \begin{matrix} 5.500 & 1.400 & \\ 0.660 & 0.697 & \text{-> toxic,} \end{matrix}$$

$$\boldsymbol{p}_1 = \begin{matrix} 5.220 & 0.417 & \\ 0.880 & 0.571 & \text{-> nontoxic.} \end{matrix}$$

Meanwhile, M performs its learning on the modified training set T'':

blue	sometimes	->	toxic,
green	always	->	toxic,
green	always	->	toxic,
blue	never	->	nontoxic,
blue	sometimes	->	toxic,
red	never	->	nontoxic,
green	sometimes	->	nontoxic,
blue	sometimes	->	nontoxic,
green	always	->	nontoxic.

Now, when an unknown instance $\boldsymbol{x}$ = (5.0, blue, sometimes, 1.1) is presented to the system, classification proceeds as follows. SP/MSP takes the modified instance $\boldsymbol{x}'$ = (5.0, 1.1) and classifies it using $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$. In this case,

$$d_0 = \frac{|5 - 5.5|}{.66} + \frac{|1.1 - 1.4|}{.697} = 1.19,$$

$$d_1 = \frac{|5 - 5.22|}{.88} + \frac{|1.1 - .417|}{.571} = 1.45.$$

Since $d_0 < d_1$, SP/MSP classifies $\boldsymbol{x}'$ as toxic. However, since this is a tentative classification that uses only part of the available information, SP/MSP passes its guess along to M which uses this information along with the modified instance x'' = (blue, sometimes) to make a final classification.

This may prove to be a viable approach even when M has the ability to process continuously valued data itself. The justification for this claim lies in the fact that many learning models are originally developed to handle nominal data and were later extended for the continuous case (cf. C4.5 and CN2). Often, the learning model is naturally better adapted to handling nominal data due to its development.

A second option for the handling of nominal data is to relinquish the use of other learning models altogether and extend SP/MSP to handle the nominal case itself. This approach has potential since some work has been done in the area of handling multiple input domains [Aha91][Gir94][Sta86]. However, by the same reasoning used in employing SP/MSP as a specialized processor for continuously valued attributes, it may be argued that some learning method that "naturally" handles nominal values should be employed as a specialized processor for handling nominally valued attributes.

4.6. Parallel Implementation of SP/MSP

Parallel implementation of an algorithm can be of benefit if it results in significant speedup. In the case of the MSP algorithm, this is a necessity. $O(n^3)$ is an expensive time requirement for training sets of greater than moderate size. A c-ary tree with a broadcast and gather scheme (algorithm 4.10) is therefore presented as a parallel implementation of the MSP algorithm.

Create a c-ary tree of one level by creating a single node that represents the initial prototype for each output class as in Figure 4.7. The root node contains the training set and each prototype node contains the vector of means and standard deviations for the class it represents.

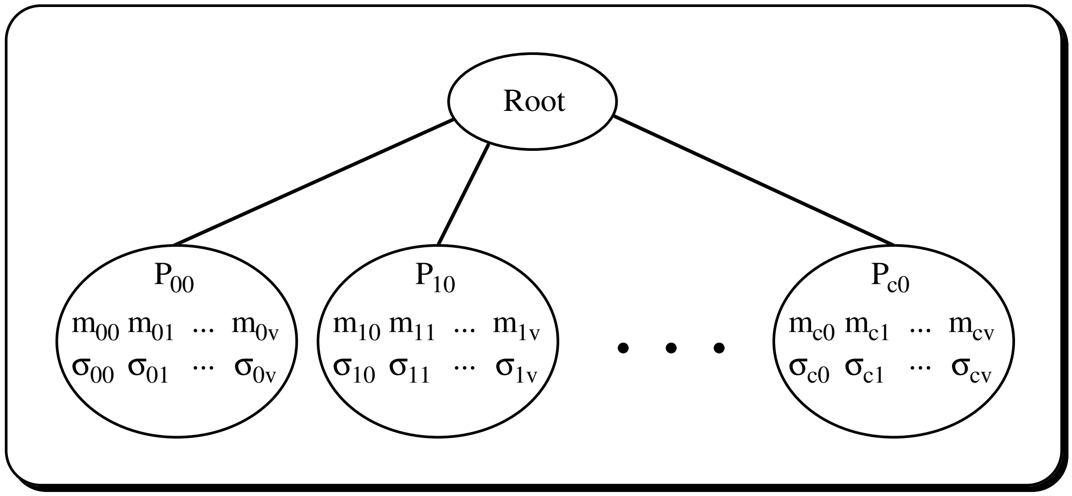

Figure 4.7. Initial c-ary tree after running SP

Incidentally, the single level c-ary tree shown in Figure 4.7 is the entire parallel structure if SP were to be implemented. However, since SP's complexity is already $O(n)$, nothing is gained from a parallel implementation, at least in terms of speedup.

Calculation of ε_b can now be performed as the root node broadcasts each instance to the prototype nodes and they in turn return their distance function d_i. The root node then finds the minimum d_i and determines whether the classification is correct. Next, each prototype node creates v subtrees (one for each of its input variables) of three temporary nodes. See Figure 4.8.

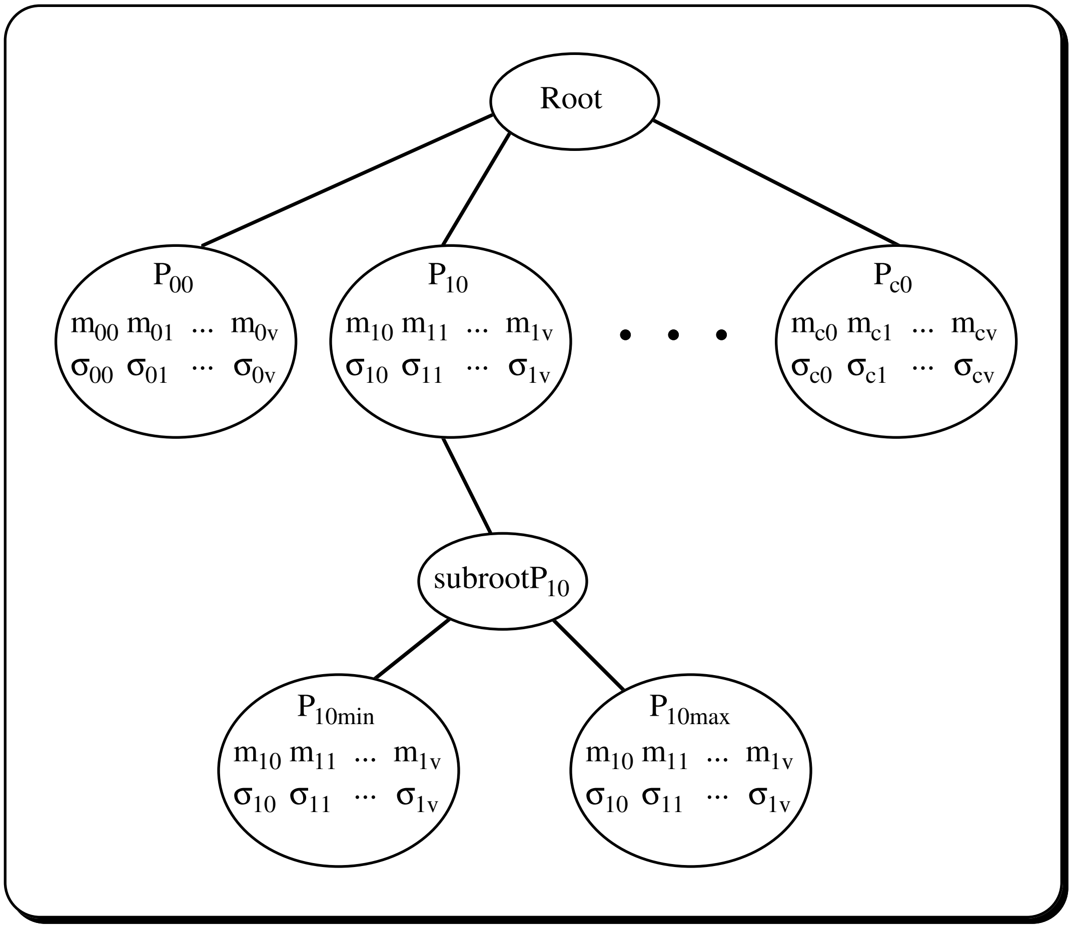

Figure 4.8. A temporary subtree for one input of one prototype

All ε_a are then calculated in parallel as follows. The root again broadcasts each instance to each prototype node and each prototype node again sends its d_i back to the root. But each prototype node also forwards the broadcast instance to each of its temporary subtrees. These contain two temporary prototypes which calculate their distance functions d_{imin} and d_{imax} and send them to the subroot node above. Also, this time the d_i sent up to the root node from the permanent prototypes are forwarded from the root to all the subroot nodes. The subroot nodes can then find the minimum. This is repeated over all instances as each subroot node calculates its particular version of ε_a. Finally, all these ε_a are gathered by the root and the smallest one, ε_{small}, is compared to ε_b. If $\varepsilon_{small} < \varepsilon_b$, the root sends a message to the prototype whose subtree generated ε_{small} and that prototype is permanently replaced by the two leaves of the subtree responsible (Figure 4.9). These two new prototypes then create their own subtrees and the entire process is repeated. When eventually $\varepsilon_{small} \geq \varepsilon_b$, the root broadcasts a message to all prototypes to destroy their temporary subtrees and the process is complete.

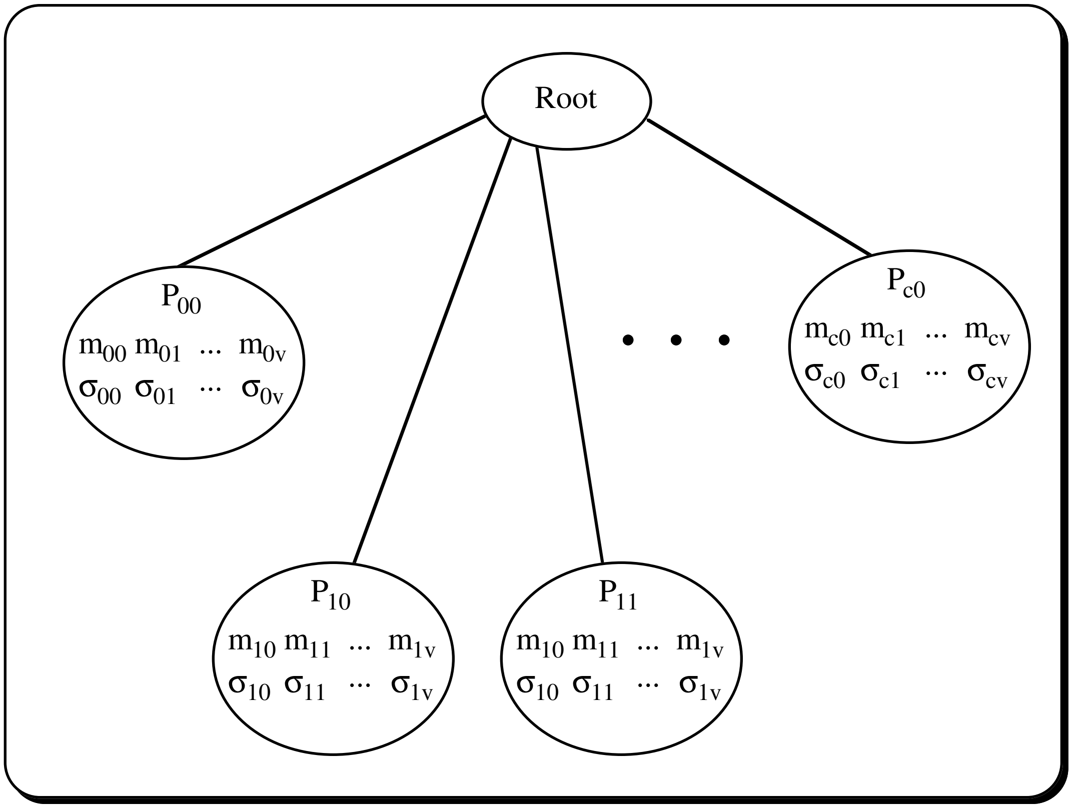

Figure 4.9. Prototype for class o_1 has been split into two new prototypes

4.6.1. *Analysis.* Analysis of the above algorithm yields $O(nv)$ steps to create the initial prototype nodes. To create a new prototype then requires $O(nv)$ steps to calculate ε_b, $3v$ steps to create the temporary nodes, $nvlog_cn$ steps to calculate their temporary means and standard deviations, and $cnvlog_cn$ steps to classify the training set and calculate ε_{small}. The factor of c in the last term is due to the root node's gathering of all d_i and broadcasting them to all the subroots for each instance. To create a new prototype, then, requires $O(nv + 3v + nvlog_cn + cnvlog_cn) = O(nlog_cn)$. Unfortunately, the number of new prototypes to be created is still bounded above by n and thus the entire parallel algorithm still requires $O(n^2)$ steps. However, although the theoretical upper bound on the algorithm is n^2 due to the fact that the number of prototypes created is bounded above by n, it is encouraging to note that empirically (that is, for the average case) that number is much lower (see Table 4.6) so that empirically the parallel algorithm begins to approach the $O(nlogn)$ expected from an implementation based on a tree topology.

```
Parallel Learn()
    create root node that contains training set
    run algorithm 4.1 to create c prototype nodes
    calculate e_b
        root broadcasts each instance
        prototype node i calculates d_i and sends result to root
        root determines closest
    Create Subtrees()
    calculate all ε_a
        root broadcasts each instance
        prototype node i calculates d_i and sends result to root
        root forwards all d_i to each prototype node
        prototype node i sends instance and all d_i to subroot
        subroot broadcasts instance to max and min
        max and min calculate d_imax and d_imin and send results to subroot
        each subroot determines closest
    all ε_a are gathered by root and ε_small is found
    if ε_small < ε_b
        replace prototype node that produced ε_small  with leavesof its subtree
        Create Subtrees()
        Parallel Learn()

Create Subtress()
    for all prototype nodes without a subtree
        create v children called subroots (one for each input variable)
        for each subroot create 2 children -- max and min
            max represents instances closest to max value of variable v
            min represents instances closest to min value of variable v
```

Algorithm 4.10. Parallel implementation of MSP

Chapter 5
Conclusion and Future Research

This chapter discusses ongoing research and areas for further investigation as well as summarizing the work presented in this thesis.

5.1. Future Research

As has been suggested throughout the preceding chapters, many fruitful avenues of inquiry still exist. These include but are not limited to using the distance function d as a confidence measure during classification, improving the metric for determining the number of prototypes in MSP (not only to improve performance but also to decrease algorithmic complexity), combination of SP/MSP with a learning system M that handles nominal data, and/or extension of SP/MSP to handle nominal data. Each of these areas is briefly discussed below.

5.1.1. *Using d as a Confidence Measure*. Confidence measures are often used in learning. In many cases it is not only good to know the prototype closest to the input $\boldsymbol{x}$, but also to know how close this prototype is in comparison with others. If the prototype is much closer than any of the others, the system may be more confident that it has made a correct classification. On the other hand, if several prototypes are almost equally near $\boldsymbol{x}$, the system must be less sure it has guessed correctly. The distances d_i lend themselves nicely to the use of measuring confidence in classification correctness. One possibility is to use a weighted sum of the ratios of the closest prototype to the remaining prototypes--

$$confidence = 1 - \left(\sum_{loser=0}^{P} \frac{d_{winner}}{d_{loser}} \right), loser \neq winner,$$

where $P = \sum_{i=0}^{c-1} q_i$ = the total number of prototypes. Other methods of measuring confidence may include use of the variances or covariances associated with each prototype.

5.1.2. *Improved Metric for Number of Prototypes*. It has already been suggested that an improved metric for determining how many prototypes to create could significantly improve performance on some data sets. This is not a trivial problem however, as it is essentially the same as determining the modality of the data. Progress in this area could improve MSP's performance, and may also reduce its computational complexity as well. This approach could also be combined with the idea of confidence discussed above. Care must be taken, however, not to improve this metric "too much" in the sense that it is certainly possible to improve modeling of the training set (using Bayesian statistics, for example). However, modeling the training set too well leads to overfitting the data and thus to poor generalization.

5.1.3. *Combining SP/MSP with M*. This issue has already been discussed briefly, however, it is not a trivial one. For example, some issues that need to be addressed include 1) Should SP/MSP precede *M* or vice versa? 2) Or is it problem dependent? 3) If SP/MSP is allowed to handle all continuously valued data while *M* handles all nominally valued data, obviously some correlations are still going to be destroyed. Is this unavoidable? 4) How should the information produced by SP/MSP be combined with that produced by *M*?

For example, an alternative approach to the example in 4.5.1 is to use SP/MSP as a discretizer only and not allow it to directly help classify new instances. One way to do this is once prototypes are obtained using T', classify T' using these prototypes and then recombine this information into T'' before using T'' to train *M*. Again using the example from 4.5.1, T' is

6.0	1.8	->	toxic,
6.3	1.9	->	toxic,
4.9	1.7	->	toxic,
4.6	0.2	->	nontoxic,
4.8	0.2	->	toxic,
5.3	0.2	->	nontoxic,
5.1	1.1	->	nontoxic,
5.2	1.4	->	nontoxic,
5.9	1.5	->	nontoxic,

and

$$\boldsymbol{p}_0 = \begin{matrix} 5.500 & 1.400 & \\ 0.660 & 0.697 & \text{-> toxic,} \end{matrix}$$

$$\boldsymbol{p}_1 = \begin{matrix} 5.220 & 0.417 & \\ 0.880 & 0.571 & \text{-> nontoxic.} \end{matrix}$$

Now, using $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$, calculate distances from all prototypes for each instance:

	d_0	d_1
(6.0, 1.8)	1.33	3.31
(6.3, 1.9)	1.93	3.82
(4.9, 1.7)	1.34	2.61
(4.6, 0.2)	3.09	1.08
(4.8, 0.2)	2.78	0.86
(5.3, 0.2)	2.02	0.47
(5.1, 1.1)	1.04	1.33
(5.2, 1.4)	0.45	1.74
(5.9, 1.5)	0.75	2.70.

Then, reclassify each instance in T'. In this case:

6.0	1.8	->	toxic,
6.3	1.9	->	toxic,
4.9	1.7	->	toxic,
4.6	0.2	->	nontoxic,
4.8	0.2	->	nontoxic,
5.3	0.2	->	nontoxic,
5.1	1.1	->	toxic,
5.2	1.4	->	toxic,
5.9	1.5	->	toxic.

Finally, construct T'' using this information by replacing the continuously valued input in each instance with the tentative classification produced by SP/MSP:

toxic	blue	sometimes	toxic	->	toxic,
toxic	green	always	toxic	->	toxic,
toxic	green	always	toxic	->	toxic,
nontoxic	blue	never	nontoxic	->	nontoxic,
nontoxic	blue	sometimes	nontoxic	->	toxic,
nontoxic	red	never	nontoxic	->	nontoxic,
toxic	green	sometimes	toxic	->	nontoxic,
toxic	blue	sometimes	toxic	->	nontoxic,
toxic	green	always	toxic	->	nontoxic.

T'' now consists of all nominally valued data and is used to train M. Classification is performed on $\boldsymbol{x}$ = (5.0, blue, sometimes, 1.1) by first using the prototypes of SP/MSP to discretize the instance so $\boldsymbol{x'}$ = (toxic, blue, sometimes, toxic). M then outputs the classification it finds for $\boldsymbol{x'}$.

5.1.4. *Extending SP/MSP to Handle Nominal Data*. Several approaches may be tried here. First, some method of converting nominally valued data into a form for which the statistical approaches of SP/MSP have some relevance may be found. The success of this approach, of course, depends on how faithfully the conversion adapts the nominal data to a continuously valued domain while preserving the relationships they represent. A second approach would be to modify the distance function d so that it acts statistically in the continuously valued case and in a hamming distance (or related method) type manner in the nominal case. A third approach might be to attempt to develop some M specifically to act as a partner to SP/MSP. In doing so, customized interaction of SP/MSP and M may overcome some of the difficulties that will be encountered in attempting to integrate SP/MSP and some M' not specifically designed to work with SP/MSP.

5.2. Conclusion

The ability to handle continuously valued data is crucial to the success of any general purpose learning algorithm. Discretization of the data, in essence constraining all problems to consist solely of nominal (or at best linearly ordered) values, is one possibility.

Chapter 2 presents a new approach to discretization called Boundary Ranking and Classification Evaluation (BRACE). BRACE is an eclectic paradigm that seeks to incorporate advantages from many other discretization methods. Two example algorithms, VALLEY and SLICE, when compared with other discretization methods demonstrate the utility of BRACE as a discretization preprocessor on several real-world problems.

The main results in chapter 3, however, indicate that in general, the choice of which discretization method to use depends both on the problem to be learned as well as on the choice of supervised learning algorithm. As a result, none of the discretization methods clearly out-performed the others, although ChiMerge appears to be somewhat more robust than the other methods. Also, the process of discretization as a preprocessing step suffered significantly when compared with algorithms that directly handle continuously valued data. Each of the supervised learners that employed preprocessing performed poorly on several of the problems, even assuming choice of the best discretization method for the combination of problem and learner.

As a result, chapter 4 takes a different approach and develops a method of learning from continuously valued data. The main ideas are presented in a simple model, statistical prototypes (SP). This model is then generalized to multiple statistical prototypes (MSP). MSP compares favorably with other learning algorithms, outperforming all of them on six of eleven of the data sets, and performing average or better on 9 of 11 data sets. These promising empirical results indicate that MSP is a viable approach to learning from continuously valued data, even with a very simple metric for determining the number of prototypes per class. Further experimentation has indicated that MSP may be further improved by improving this metric. Because MSP was specifically designed to address the problem of learning from continuously valued attributes, it does not in its present form handle nominal data. Thus, by attempting to solve the problem of how to handle continuously valued data, the inverse problem of how to handle nominal data has been created.

Chapter 5 addresses this issue somewhat and explores several avenues of ongoing research that hold promise. The eventual goal should be a synergistic approach that does not ignore the unique challenges of either kind of data but rather attempts to take advantage of the salient features of both.

This research was supported in part by grants from Novell, Inc. and WordPerfect Corporation.

Bibliography

[Aha91] Aha, D.W.; Kibler, D.; and Albert M.K, "Instance-Based Learning Algorithms", *Machine Learning*, v6, Kluwer Academic Publishers, The Netherlands, pp.37-66, 1991.

[Aha89] Aha, D. W., "Incremental, instance-based learning of independent and graded concept descriptions", *Proceedings of the Sixth International Workshop on Machine Learning*, pp. 387-391, 1989.

[And93] Andersen, Timothy L. and Martinez, Tony R., "Learning and Generalization with Bounded Order Critical Feature Sets", *Proceedings of the 6th Australian Joint Conference on Artificial Intelligence*, p. 450, 1993.

[Bre93] Breiman, Leo; Friedman, Jorome; Olshen, Richard; and Stone, Charles, Classification and Regression Trees, Chapman and Hall, New York, 1993.

[Cat91] Catlett, J., "On Changing Continuous Attributes Into Ordered Discrete Attributes", *Lecture Notes in Artificial Intelligence*, Ed. J. Siekmann, Springer-Verlag, Berlin, pp. 164-78, 1991.

[Cho72] Chow, C.K. and Kaneko, T., "Automatic Boundary Detection of the Left Ventricle from Cineangiograms", *Computers and Biomedical Research*, v5, pp. 388-410, 1972.

[Cla89] Clark, Peter and Niblett, Tim, "The CN2 Induction Algorithm", *Machine Learning*, v3, Kluwer Academic Publishers, The Netherlands, pp. 261-83, 1989.

[Das91] Dasarathy, Belur, Nearest Neighbor Norms: NN Pattern Classification Techniques, IEEE Computer Society Press, Los Alamitos, California, 1991.

[Doy62] Doyle, W., "Operations Useful for Similarity-Invariant Pattern Recognition", *Journal of Associative Computing Machinery*, v9, pp. 259-67, 1962.

[Eve74] Everitt, Brian, Cluster Analysis, Halsted Press, New York, 1974.

[Fay92] Fayyad, Usama M. and Irani, Keki B., “On the Handling of Continuous-Valued Attributes in Decision Tree Generation”, *Machine Learning*, v8, pp. 87-102, 1992.

[Fis90] Fisher, Douglas H., "Knowledge Acquisition Via Incremental Conceptual Clustering", Readings in Machine Learning, eds. Jude W. Shavlik and Thomas G. Dietterich, Morgan Kaufman Publishers, Inc., San Mateo, California, pp. 267-83, 1990.

[Gen90] Gennari, John H.; Langley, Pat; and Fisher, Doug, "Models of Incremental Concept Formation", Machine Learning: Paradigms and Methods, ed. Jaime Carbonell, MIT press, Cambridge, Massachusetts, pp. 11-61, 1990.

[Gir94] Giraud-Carrier, C. and Martinez, T., “An Efficient Metric for Heterogeneous Inductive Learning Applications in the Attribute-Value Language”, To appear in *Proceedings of the Third Golden West International Conference on Intelligent Systems*, 1994.

[Gon87] Gonzalez, Rafael C. and Wintz, , Digital Image Processing, Addison-Wesley Publishing Company, Reading, Massachusetts, pp. 331-88, 1987.

[Hay94] Haykin, Simon, Neural Networks: A Comprehensive Foundation, Macmillan College Publishing Company, New York, 1994.

[Jai88] Jain, Anil K. and Dubes, Richard C., Algorithms for Clustering Data, Prentice Hall, New Jersey, 1988.

[Jar73] Jarvis, R. A. and Patrick, Edward A., "Clustering Using a Similarity Measure Based On Shared Nearest Neighbors", *IEEE Transactions on Computers*, vC-22, no. 11, pp. 1025-34, 1973.

[Ker92] Kerber, Randy, "ChiMerge: Discretization of Numeric Attributes", *Proceedings of the 10th National Conference on Artificial Intelligence*, pp. 123-7, 1992.

[Leb85] Lebowitz, Michael, “Categorizing Numeric Information for Generalization”, *Cognitive Science*, v9 no. 3, pp. 285-308, 1985.

[Mar91a] Martinez, T. R. and Campbell, D. M., "A Self-Organizing Binary Decision Tree for Incrementally Defined Rule Based

Systems", *IEEE Transactions on Systems, Man, and Cybernetics*, v21, no. 5, pp. 1231-8, 1991.

[Mar91b] Martinez, T. R. and Campbell, D. M., "A Self-Adjusting Dynamic Logic Module", *Journal of Parallel and Distributed Computing*, v11, no. 4, pp. 303-13, 1991.

[Mar90] Martinez, Tony R., "Adaptive Self-Organizing Concurrent Systems", *Progress in Neural Networks,* vol. 1, ch. 5, ed. O. Omidvar, Ablex Publishing, pp. 105-26, 1990.

[Mar88] Martinez, T. R. and Vidal J. J., "Adaptive Parallel Logic Networks", *Journal of Parallel and Distributed Computing*, v5, no.1, pp. 26-58, 1988.

[Mar86] Martinez, T. R., "Adaptive Self-Organizing Logic Networks", Ph.D. Dissertation, UCLA Technical Report- CSD 860093, June, 1986.

[Mic90] Michalski, Ryszard S., "A Theory and Methodology of Inductive Learning", Readings in Machine Learning, eds. Jude W. Shavlik and Thomas G. Dietterich, Morgan Kaufman Publishers, Inc., San Mateo, California, pp. 70-95, 1990.

[Mic83] Michalski, Ryszard S. and Stepp, Robert, E., "Learning From Observation: Conceptual Clustering", Machine Learning: An Artificial Intelligence Approach, eds. Ryszard S. Michalski, Jaime G. Carbonell, Tom M. Mitchell, Tioga Publishing Company, Palo Alto, California, pp. 331-63, 1983.

[Mur92] Murphy, P. M. and Aha, D. W., *UCI Repository of machine learning databases*. Irvine, CA: University of California, Department of Information and Computer Science, 1992.

[Qui93] Quinlan, J. Ross, C4.5: Programs For Machine Learning, Morgan Kaufman Publishers, San Mateo, California, 1993.

[Qui90] Quinlan, J. R., "Induction of Decision Trees", Readings in Machine Learning, eds. Jude W. Shavlik and Thomas G. Dietterich, Morgan Kaufman Publishers, Inc., San Mateo, California, pp. 57-69, 1990.

[Ris83] Rissanen, Jorma, "A Universal Prior for Integers An Estimation by Minimum Description Length", *Annal of Statistics*, v11 no. 2, pp. 416-31, 1983.

[Rum86] Rumelhart, D. E.; McClelland, J. L., Parallel Distributed Processing, MIT Press, chs. 5, 8, pp.151-193, 318-62, 1986.

[Sch87] Schlimmer, Jeffrey C., "Learning and Representation", *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI87)*, v2, pp.511-515, 1987.

[Spa80] Spath, Helmuth, Cluster Analysis Algorithms, Ellis Horwood Limited, West Sussex, England, 1980.

[Sta86] Stanfill, C. and Waltz, D., "Toward Memory-Based Reasoning", *Communications of the ACM*, v29, no. 12, pp. 1213-28, 1986.

[Tou74] Tou, J. T. and Gonzalez, R. C., Pattern Recognition Principles, Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

[Ven94] Ventura, Dan and Martinez, T. R., "BRACE: A Paradigm For the Discretization of Analog Data", *Proceedings of the Seventh Florida Artificial Intelligence Research Symposium*, pp. 117-21, 1994.

[Wes78] Weska, J.S., "A Survey of Threshold Selection Techniques", *Computer Graphics Image Processing*, v7, pp. 259-65, 1978.

[Whi83] White, J.M. and Rohrer, G.D., "Image Thresholding for Optical Character Recognition and Other Applications Requiring Character Image Extraction", *IBM Journal on Research and Development*, v27, no. 4, pp. 400-11, 1983.

[Wid88] Widrow, Bernard and Winter, Rodney, "Neural Nets for Adaptive Filtering and Adaptive Pattern Recognition", *IEEE Transactions on Computers,* v37, no. 3, 1988.

[Won87] Wong, A. K. C., and Chiu, D. K. Y., "Synthesizing statistical knowledge from incomplete mixed-mode data", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vPAMI-9, no. 6, pp.796-805, November 1987.

[Zar94] Zarndt, Frederick, "An Empirically Derived Test Suite for Machine Learning and Connectionist Algorithms", in preparation, 1994.

On Discretization as a Preprocessing Step
For Supervised Learning Models

Dan Ventura

Department of Computer Science

M.S. Degree, April 1995

ABSTRACT

Many machine learning and neurally inspired algorithms are limited, at least in their pure form, to working with nominal data. However, for many real-world problems, some provision must be made to support processing of continuously valued data. BRACE, a paradigm for the discretization of continuously valued attributes is introduced, and two algorithmic instantiations of this paradigm, VALLEY and SLICE are presented. These methods are compared empirically with other discretization techniques on several real-world problems and no algorithm clearly outperforms the others. Also, discretization as a preprocessing step is in many cases found to be inferior to direct handling of continuously valued data. These results suggest that machine learning algorithms should be designed to directly handle continuously valued data rather than relying on preprocessing or *ad hoc* techniques. To this end statistical prototypes (SP/MSP) are developed and an empirical comparison with well-known learning algorithms is presented. Encouraging results demonstrate that statistical prototypes have the potential to handle continuously valued data well. However, at this point, they are not suited for handling nominally valued data, which is arguably at least as important as continuously valued data in learning real-world applications. Several areas of ongoing research that aim to provide this ability are presented.

COMMITTEE APPROVAL: ______________________________
Tony R. Martinez, Committee Chairman

____
William A. Barrett, Committee Member

____
J. Kelly Flanagan, Committee Member

____
David W. Embley, Graduate Coordinator