Prototype Styles of Generalization

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

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

by
D. Randall Wilson
August 1994

This thesis, by D. Randall Wilson 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

Gordon E. Stokes, Committee Member

Date

David W. Embley, Graduate Coordinator

Table of Contents

List of Figures

Chapter 1
Introduction

A learning system can generalize from training set data in many ways. This thesis proposes several generalization styles using prototypes in an attempt to provide accurate generalization on training set data for a wide variety of applications. These generalization styles are efficient in terms of time and space and lend themselves well to massively parallel architectures. There is likely no one style of generalization which solves all problems, for different styles work better on some applications than others. This thesis presents a sizable collection of generalization styles, including many new ones, and demonstrates the value of using multiple styles in achieving increased generalization accuracy. Simulation results indicate that a collection of several styles can provide more accurate generalization than can any one by itself. Eventually it should be possible to construct a system which can discover intelligently which style or styles of generalization works best on any given application.

1.1. Background and Motivation

Traditional computer programming requires the programmer to define explicitly what the computer should do given any particular input. Even expert systems depend on the knowledge of experts in the field and their ability to explain how they make their decisions. When a solid knowledge of the application domain is available, this process often works quite well. For some applications, however, not enough is known about the subject to derive a formula or write a program to solve the problem, or the solution is not as flexible or accurate as desired.

Even when it is possible to write a program or consult an expert for domain knowledge, the cost of doing so may be quite high. As Quinlan (1986) put it, "While the typical rate of knowledge elucidation by [the interview approach] is a few rules per man day, an expert system for a complex task may require hundreds or even thousands of such rules. It is obvious that the interview approach to knowledge acquisition cannot keep pace with the burgeoning demand for expert systems." Programmers face a very similar problem, because many of the same rules needed for an expert system must be encoded either explicitly or implicitly as instructions in a program.

When it is not possible to write a program to solve a problem directly, or when the cost of doing so is restrictively high, it may be possible to collect a set of examples of what the output of the system should be in specific situations. Each such example is called an *instance*, and usually consists of a vector of input values (the *inputs*) representing certain features of the environment, as well as the value(s) that the system should output in response to these input values (the *output*). A collection of such instances is called a *training set*.

Much research has been done to develop systems which learn by example, or do *inductive learning*. Given a training set of instances, these systems often can "learn" the relationship between the inputs and outputs well enough to be able to receive an input and produce the correct output with a high probability, even if that input was not one of the examples. This ability is called *generalization*. Learning systems which can generalize accurately have the potential of being able to solve problems for which conventional programming solutions have not been found. They may also be able to avoid the high costs involved in writing specific programs or constructing expert systems to solve many problems. In some cases, they can perform with greater accuracy as well.

Neural networks (Lippmann 87, Rumelhart 86, Widrow 88 and 90) provide several ways to do inductive learning. Some styles of neural networks which have been successful in accurately learning to generalize from a training set, such as the backpropagation network (Rumelhart 86), have required many iterations through the learning algorithm to solve the problem, and thus have not always been able to solve problems in an acceptable amount of time. In most cases they also use several "system parameters" which must be set by hand, and this causes the learning process to be less than automatic. In addition, they sometimes suffer from the problem of getting stuck in a local minimum, so several runs of the algorithm often are necessary.

1.2. Machine Learning History and Related Work

Much work has also been done in the area of machine learning to develop inductive learning systems. The Nearest-Neighbor (NN) pattern classifier (Cover 67) keeps the entire training set of instances and generalizes by using the output of the "closest" instance to a new input. The *k*-

Nearest-Neighbor (*k*NN) algorithm uses the most common output value of the *k* closest neighbors during generalization, and performs better in the presence of noise. The Condensed Nearest-Neighbor rule (Hart 68) attempts to reduce storage requirements by keeping only selected instances, but suffers from intolerance to noise.

The nested generalized exemplars (NGE) approach uses single instances or axis-parallel hyperrectangles that may cover several instances (Salzberg 91). The NGE approach reduces storage requirements as well, but there is evidence that indicates that it may not compare favorably in terms of generalization accuracy with *k*NN in many domains (Wettschereck 94a). A hybrid method (Wettschereck 94b) improves generalization accuracy to near that of the *k*NN methods. Similar attempts have been made to combine linear separators with *k*NN methods (Dasarathy 79a and 79b).

Instance-Based Learning (IBL) algorithms (Aha 91) reduce the number of instances that must be saved while remaining tolerant of noise. They are also *incremental*, meaning that additional training instances can be added easily, even after the initial learning stage is already complete.

Case-Based Reasoning (CBR) systems (Stanfill 86, Koton 88, Kolodner 89, Porter 90) also attempt to use previously seen cases to generalize on new inputs. However, CBR systems may modify cases rather than simply storing a list of examples.

Other notable induction learning algorithms that have had some success include ID3 (Quinlan 86), which builds decision trees; C4.5 (Quinlan 93) which among other things extends ID3 to handle continuously-valued inputs and used tree pruning; and CN2 (Clark 89), which combines the AQ family of inductive algorithms (Michalski 69, 83, 86) with ID3.

The ASOCS (Adaptive Self-Organizing Concurrent System) models (Martinez 86, 90a, 91) have the attractive feature of fast (one-shot) learning. That is, they learn an instance set in one pass without the need for iteration. They also lend themselves well to simple parallel architectures which can be implemented efficiently in hardware (Rudolph 91). These features make it attractive for solving large problems very quickly, without hardware constraints.

The ASOCS models, however, expect rule-based instances (Martinez 90b), and are not designed for training set data. In addition, some ASOCS models do not perform much generalization by themselves. This research began with the goal of extending ASOCS models to perform generalization on training set data, and was extended to show the need for multiple styles of generalization.

Most of the learning models mentioned above are limited to one style of generalization which is inherent in the learning algorithm. Derivatives of the *k*NN approach are restricted in most cases to a single similarity measure or *distance metric*, which may or may not be appropriate for the given application. The model therefore works on some applications but not on others, and it often is difficult to tell beforehand when it will work, when it will not, and why.

This thesis presents a large collection of generalization styles, some of which are very similar to techniques that have been used before. Several distance metric styles of generalization similar to the *k*NN approach are used, along with some voting methods. Some new styles of generalization involving first-order features and confidence levels also are introduced. Confidence levels are used to combine first-order features with distance metrics and to weight distances and voting power, resulting in over 300 permutations of these basic generalization styles. Simulation results support the hypothesis that no single style of generalization is most suitable for all application domains. The results also indicate that the styles of generalization used in this research compare favorably to previous models in terms of generalization accuracy in several application domains.

Section 2 gives an overview of ASOCS models and addresses the issues involved in extending ASOCS models to generalize on training set data. In Section 3, the styles of generalization studied in this research are presented along with several extensions of each. Section 4 discusses a learning algorithm and mentions several implementation considerations. Section 5 gives empirical results of simulations run on a variety of applications using each of these generalization styles and gives an analysis of these results. Section 6 provides conclusions and future research areas.

Chapter 2
Extending ASOCS Models to Generalize on Training Set Data

The Adaptive Self-Organizing Concurrent System (ASOCS) models (Martinez 1986) have several advantages over other connectionist models which make them attractive as a basis for learning models. This section discusses the advantages and disadvantages of the ASOCS models, gives a brief overview of the models, and discusses how to extend the models to perform generalization on training set data.

2.1. Advantages of ASOCS Models and Needed Extensions

ASOCS models are guaranteed to learn an instance set with one presentation, without the need for iteration, and they do not get stuck in local minima, making them much faster than many other learning mechanisms. They also create a self-organizing topology and are implemented easily with simple digital logic units. This makes it possible to construct inexpensive systems which learn and execute in a massively parallel fashion.

During learning, ASOCS models expect rule-based instances with at least some of the critical variables identified by having "don't care" inputs on some of the non-critical variables. Rule-based instance sets may be useful for some applications, but require quite a bit of *a priori* knowledge about the application, and currently there are few such instance sets available for study.

Training set data, on the other hand, typically has a value asserted for all of the variables in each instance, and the order of presentation is unimportant. Training set data is much easier to obtain than rule-based instance sets, because the creation of rules requires intelligent preprocessing, *a priori* knowledge, or human intervention. Therefore, training set data is more readily available and more widely used.

Unfortunately, ASOCS models are not set up to learn standard training sets and do not usually do much generalization, especially when given standard training sets. Therefore, it would be helpful if the ASOCS models could be extended to work on training set data and generalize well, without sacrificing the advantages of fast, guaranteed learning and self-organizing massively parallel architectures.

2.2. Overview of ASOCS Models

ASOCS models operate in either learning mode or execution mode. During learning, the systems are presented with a list of rule-based instances in which some of the inputs are asserted, and others are left as "don't care" variables.

For example, given a system with four Boolean input variables A, B, C and D and one Boolean output variable *Z*, a rule written as "AD'→*Z*" would mean that when A is high (i.e., A=1), and D is low (D=0), then the output should be high (*Z*=1). In this example, B and C are "don't care" variables, because it does not matter what values they have, as long as A=1 and D=0. An equivalent way to write this rule is "1**0→1", where an asterisk ("*") indicates a "don't care" variable. This representation is used throughout this thesis for consistency with later sections.

Two instances with the same output are said to be *concordant* with respect to each other, while two with differing outputs are *discordant* with respect to each other. As an ASOCS system receives rule-based instances, it maintains a current list of rules called an *instance set*. ASOCS models guarantee to keep instance sets *consistent*. That is, they guarantee that no two discordant instances can be matched by an input vector simultaneously.

For example, the two instances "11**→0" and "**11→1" are both matched by the input vector "1111", but they are discordant, and thus assert two different output values for the same input vector. ASOCS models maintain consistency by giving one of the instances priority over the other. Usually, newer instances are given priority over older ones.

In the original ASOCS models this was done through *discriminant variable addition* (DVA), which ensures that every pair of discordant instances has at least one variable which is asserted (i.e., not "don't care") and has a different value in the two instances. For example, in the example above, the instance "11**→0" would be replaced by the two instances "110*→0" and "11*0→0" to make sure that it could never be matched at the same time as "**11→1". One disadvantage of using discriminant variable addition is that it can add many new instances to the system, especially when the inputs are multi-state instead of Boolean.

The Priority ASOCS (PASOCS) model made DVA unnecessary simply by assigning a higher priority to instances that were inconsistent with older instances in the set (Martinez 94).

ASOCS models also perform *minimization* to keep the instance set small, and to help identify critical variables. For example if the instance "11**→1" is already in the instance set, and the new instance "1***→1" is added, the old one can be removed completely, because the new instance covers all of the input space of the old one and has the same output.

A special case, called *one-difference minimization*, occurs when instances can be combined into a single instance with one variable converted into a "don't care" variable. For example, the instances "111*→1" and "110*→1" can be combined into the single instance "11**→1", since the output is the same regardless of the value of the third variable. This both reduces the size of the instance set and helps identify critical variables in the application.

2.3. Extending ASOCS Models to Handle Training Set Data

This section discusses the differences between ASOCS instance sets and training sets and show how these differences can be overcome.

2.3.1. *Differences between ASOCS Instances and Training Sets.* The first main difference between ASOCS instance sets and typical training sets is that ASOCS instances often have one or more "don't care" variables which are not asserted. These instances are assumed to cover the input space covered by all values for each such variable. In contrast, training set instances typically have a value asserted for all variables, except when that value is unknown.

Secondly, training sets often have continuously-valued variables while ASOCS instances are generally either Boolean or multi-state, and therefore can contain only a discrete number of values.

Third, ASOCS instances are learned *incrementally*, meaning that the order is important, and that inconsistencies or contradictions between instances are usually resolved by giving higher priority to later instances. In training sets, on the other hand, the order in which instances are presented is arbitrary, and should be treated as unimportant.

Fourth, the frequency of presentation of an instance may be important in training sets, while it mostly is ignored in ASOCS instance sets.

In addition, training set data more often is assumed to be "noisy," containing at times slightly wrong or even blatantly contradictory instances. Finally, they sometimes contain unknown ("don't know") input values which must be estimated, ignored or dealt with in some other manner.

Table 2.1 summarizes the differences between training sets and ASOCS instance sets.

ASOCS:	Training Set:
1. Can have "don't care" inputs	1. All inputs asserted
2. Discrete Inputs	2. Can have analog inputs
3. Order of presentation important	3. Order of presentation unimportant
4. Correct Inputs	4. Noisy and unknown inputs
5. Consistency Enforced	5. Inconsistent data
6. Frequency Ignored	6. Frequency important

Table 2.1. Comparison of ASOCS instance sets and training sets.

2.3.2. *Overcoming the Differences.* To handle the lack of "don't care" inputs, all instances from the training set can be thought of simply as ASOCS instances with all the variables asserted. This does, however, make the need for generalization more important than ever, for ASOCS instances with "don't care" variables potentially can cover large portions of the input space, while instances with all variables asserted only cover one atomic "cell" of the input space. This means that the vast majority of possible input vectors will have undefined outputs unless some additional generalization scheme is added.

To handle continuously valued variables, the training set can be preprocessed to discretize the data, in any of a large number of ways. One simple method, called the *equal width intervals* method, is to divide the range of each input variable into n even partitions, and then use the values 1 through n to represent any continuously valued variables within that partition. Another simple scheme, called the *equal-frequency intervals* method, is to do a histogram of the distribution of values for each continuously valued variable and then divide the distribution among n equally-sized blocks.

To overcome the incremental learning style used by ASOCS, the system can be allowed to retain all of the instances—even inconsistent or contradictory ones. In addition, a measure of how often an instance is contradicted is kept, along with a measure of how often the instance was supported. These measures can help identify noisy instances and retain information about the frequency of instances as well.

Instances with unknown inputs can have unknown values marked with a special value, represented by a question mark ("?"), and these values can be dealt with in a variety of ways during generalization. Each unknown input can be considered a "mismatch" with any other input (including another unknown input), or can be treated as just another input value. In that case an unknown input will be a mismatch with anything except another unknown input. Alternatively, it could be treated as a "don't care" variable, so that it matches anything, or it could be dealt with in other ways. More is said on this subject in Section 4.5.

2.3.3. *Adding Generalization.* In the original ASOCS models, much of the input space is covered by rules with many "don't care" input values. The remainder of the input space does not have an output value defined for it, however, so the system must either output a default value, as is done in Adaptive Algorithm 3 (AA3), admit it does not know, as in AA2, or attempt to generalize, as is done in AA1.

As mentioned above, when every instance in the training set has all input variables asserted, most of the input space remains uncovered, and the need for generalization becomes immensely important.

The next section describes several styles of generalization that can be used with the modified version of the ASOCS models given above. These styles retain the advantages of fast, guaranteed learning and good support of parallel techniques. They also add the ability to learn and generalize on training set data with fairly good accuracy. These styles of generalization include various distance metrics, voting schemes, first-order feature detection, and combinations of these methods.

Chapter 3
Prototype Styles of Generalization

The research presented in this thesis makes use of several generalization styles which use prototypes in an attempt to provide accurate generalization in an efficient and potentially parallel manner. *Prototypes* are similar to instances in the training set, but may represent several instances with the same output and can have additional features. When prototypes are used, they are compared with new inputs to determine the output. In such approaches, generalization styles determine both how the prototypes are created and how they are used (Wilson 93b). In this research prototypes consist of the following.

- A vector of integer input values representing features in the environment (*inputs*) and one or more output values, representing the supposedly correct response of the system (*output*).
- A count of how many instances the prototype represents, called the *frequency*.
- The *conflict*, a count of how many instances had the same input vector as the prototype, but a different output class.
- A ratio of *frequency* to *frequency* + *conflict*. This gives a probability that the inputs of the prototype should map to its output(s), and is called the *integrity*.

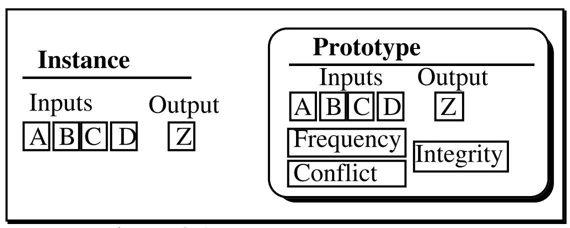

Figure 3.1. Instance vs. Prototype

Above is an illustration of an instance in a training set and a prototype.

3.1. Advantages & Disadvantages of Prototype Generalization

The prototype styles of generalization presented in this thesis have several attractive features.

- They can learn a training set quickly (usually in O($n\log n$) time, where n is the number of instances in the training set)
- They provide good generalization
- They are simple and intuitive
- They lend themselves well to parallel architectures (e.g. a binary tree)
- They do not suffer from exponential storage or processing problems.

On the other hand, they do store most of the instances in the training set, which can be sizable for large training sets. Also, since they do not throw away any information, they may "overlearn" the training set instead of picking out only the most relevant attributes of an application. This can result in a susceptibility to noise and reduced generalization accuracy.

3.2. Definitions and Notation

A *Training Set*, T, is an unordered collection of t *instances*, which are not necessarily unique. Each *instance*, or *training example*, E, has an *input vector*, $\boldsymbol{I}$, of integers, and an integer *output value*, Z. An input vector is an array of m *input values*, I_1 through I_m, where m is the number of input variables in the application represented by the training set. (Some applications have more than one output variable as well, but in this research the output variables are not considered individually, and are treated as an atomic unit.)

The components of instances, prototypes or features are written using a dot notation when it is important to distinguish which object a component belongs to. For example, $E.Z$ refers to the output Z of an instance E.

Each variable A has a *cardinality*, written $|A|$, which is the number of different values A can have. These values are numbered 0 through $|A|$-1, and cardinality sometimes is expressed by calling A a(n) $|A|$-state variable. For example, a Boolean variable would have a cardinality of two, and can be called a two-state variable. The cardinality of the i^{th} input variable is written $|A_i|$.

In addition to the values 0 through $|A|$-1, a variable can also have the value "don't care", represented by an asterisk ("*"), or "don't know" (unknown), represented by a question mark ("?").

A *nominal* variable is one for which the values 0 through $|A|$-1 have no linear meaning (e.g., color = red, green, blue, or brown). A *linear* variable is one for which the finite set of values represent some linear ordering (e.g., size=small, medium or large). A *continuously-valued* variable can have an infinite number of different values along a continuum (such as temperature or weight). In this research, continuously-valued variables are always *discretized* into a linear variable with a finite cardinality. This process is discussed in Section 4.4.

Two input values a and b are said to be *equal* if and only if they are exactly the same, i.e., they are both "*", both "?", or both the same number. On the other hand, a is said to *match* b if

(1) they are exactly the same number, or
(2) at least one of them is "*".

Note that if either or both values are "?", they do not match (See Section 4.5 for a discussion on unknown input values).

Similarly, two input vectors $\boldsymbol{A}$ and $\boldsymbol{B}$ are *equal* if and only if

$$\boldsymbol{A}_{\mathrm{i}} = \boldsymbol{B}_i \text{ for all } i,\ 1 \le i \le m,$$

and two input vectors $\boldsymbol{A}$ and $\boldsymbol{B}$ *match* if and only if

$$\boldsymbol{A}_i \textit{ matches } \boldsymbol{B}_i \text{ for all } i,\ 1 \le i \le m.$$

A *prototype*, P, has an input vector, I, and an output value, Z, as well as a *frequency*, f, a *conflict*, c, and an *integrity*, g, as explained above in Section 3.1. The frequency of a prototype, $P.f$, is defined as the number of instances E in T for which $E.\boldsymbol{I}=P.\boldsymbol{I}$ and $E.Z=P.Z$. A prototype's conflict, $P.c$, is defined as the number of instances E in T for which $E.\boldsymbol{I}=P.\boldsymbol{I}$ but $E.Z \neq P.Z$. The integrity is defined as the ratio of f to c, i.e.,

$$g = \frac{f}{f+c}$$

A *feature* has the same components as a prototype (namely, $\boldsymbol{I}$, Z, f, c and g). The order of a feature is the number of variables which are *asserted* (i.e., set to one of the values 0 through $|A|$-1) in its input vector. Thus, a *first-order feature*, F, has the restriction that one and only one of its input variables is asserted. All other input variables for that feature are set to "*" (i.e., "don't care"). First-order features do not contain unknown ("?") inputs.

3.3. Using Multiple Styles

It is believed that no single style of generalization will provide adequate generalization on all applications, so several styles are used in this research. This suite of generalization styles includes several distance metrics, some critical variable detection algorithms, voting schemes, and combinations of these. These styles are listed in Figure 3.2 and discussed below in sections 3.4-3.11.

Distance Metrics	Hamming distance Hamming distance divided by number of states Sum of absolute differences Sum of absolute differences divided by #states Euclidean distance
First-order Features	Single feature with highest frequency Single feature with highest integrity Single feature with highest global corellation Best feature for each variable vote All matching features vote
Confidence Levels	CL=Prototype's integrity CL=Prototype's best feature's integrity CL=Average integrity of prototype's features CL=Weighted overall integrity of its features Squashing function on each of these CL=CL*CL for each of these

Figure 3.2. Generalization styles used in this research.

3.4. Distance Metrics

Distance metrics are used to find out which prototype or prototypes are "closest" to the new input vector in question. The output value of the closest prototype or group of prototypes is used for the output value in response to the new input. There are several distance metrics which can be used to determine how "close" an input is to one of the prototypes. There are also several voting functions which can be used to aid in selecting which output to use, given a group of "close" prototypes.

The overall algorithm used to generalize using distance metrics is given below in Algorithm 3.1. Alternatives for the function "DIST" are defined as each distance metric is introduced, and the function "VOTE" is defined in Section 3.5. More alternatives for the VOTE and DIST functions are presented in sections 3.8 and 3.9, respectively, which discuss confidence levels.

Algorithm 3.1. Distance Metric Generalization

Definitions

DIST($\boldsymbol{A}$, $\boldsymbol{B}$) = the distance between the two input vectors $\boldsymbol{A}$ and $\boldsymbol{B}$. (The definition of this function depends on the distance metric being used.)

k = the minimum number of prototypes used to determine the output.

VOTE(S, z) = the number of votes that the prototypes in the set S cast for the output value z. (The definition for this function depends on the form of voting being used.)

Execution:

Given a set of prototypes P_1 through P_n and an input vector $\boldsymbol{I}$ to be classified:

1. For each prototype P_i, $1 \leq i \leq n$
   d = DIST($P_i.\boldsymbol{I}$, $\boldsymbol{I}$)
2. Find Min, a set of the k prototypes with minimum values for d, along with any prototypes whose d equals the k^{th} smallest d.
3. For each output value z appearing in any prototype in Min,
   v_z = VOTE(Min, z)
4. Set the output to the value z for which v_z is greatest.
   If there is a tie, use an alternate voting function to break the tie.

Five alternatives for the DIST function are presented below.

3.4.1. *Hamming Distance.* Hamming Distance (HD) is an intuitive metric, and is simply the number of input variables which do not match the prototype, i.e.,

$$\mathrm{HD}(\boldsymbol{A},\boldsymbol{B})=\sum_{i=1}^{m} match(\boldsymbol{A}i,\boldsymbol{B}_i), \qquad \text{(Eq. 3.1)}$$

where *match(a,b)* is defined as 1 if *a* matches *b*, or 0 otherwise. Also, *m* is the number of input variables in the input vectors for an application. The prototype with the least number of mismatched variables has the smallest distance using this metric.

3.4.2. *Absolute Difference.* When values represent a linear value in multi-state (non-Boolean) variables, the closeness of the value itself often helps to indicate the "distance" of an input variable to a variable in an instance. For example, if a variable such as *age group* has 10 possible values, a difference of 1 for that variable would be less than a difference of 8, while the Hamming Distance would be 1 in either case. The Absolute Difference (AD) metric is defined as:

$$\mathrm{AD}(\boldsymbol{A},\boldsymbol{B}) = \sum_{i=1}^{m} |\boldsymbol{A}_i - \boldsymbol{B}_i|, \qquad \text{(Eq. 3.2)}$$

3.4.3. *Normalized Absolute Difference.* An extension of the Absolute Difference scheme is to *normalize* the distances by dividing the distance for each variable by its cardinality. For example, a difference of 1 for a Boolean variable would be 1/2, while a difference of 1 for a 10-state variable would be 1/10. The Normalized Absolute Difference (NAD) is:

$$\mathrm{NAD}(\boldsymbol{A},\boldsymbol{B}) = \sum_{i=1}^{m} \frac{|\boldsymbol{A}_i - \boldsymbol{B}_i|}{card(i)}, \qquad \text{(Eq. 3.3)}$$

where *card(i)* is the cardinality of the i^{th} variable.

3.4.4. *Normalized Hamming Distance* Similarly, the Hamming distance itself could be divided by the number of states, since being mismatched on a 10-state variable might be considered less "wrong" than being mismatched on a Boolean one. The Normalized Hamming Distance (NHD) is defined as:

$$\mathrm{NHD}(\boldsymbol{A},\boldsymbol{B}) = \sum_{i=1}^{m} \frac{match(\boldsymbol{A}_i,\boldsymbol{B}_i)}{card(i)} \qquad \text{(Eq. 3.4)}$$

3.4.5. *Euclidean Distance.* Euclidean Distance (ED) is similar to the Absolute Difference scheme, except that the individual differences are squared before they are summed, i.e.,

$$\mathrm{ED}(\boldsymbol{A},\boldsymbol{B}) = \sqrt{\sum_{i=1}^{m} |\boldsymbol{A}_i - \boldsymbol{B}_i|^2} \qquad \text{(Eq. 3.5)}$$

In practice the final square root can be left out in order to speed up processing since the goal is to find the "closest" prototype, while the actual distance is unimportant. The Absolute Difference and Euclidean Distance measures have been called the Minkowskian metric with r=1 and r=2, respectively (see Duda 73). The Euclidean Distance often is used in the k-Nearest Neighbor algorithm and its derivatives as well. However, many variations involving confidence levels and voting are presented in later sections.

Example 3.1. Consider a training set with four input variables, the first three of which are Boolean, and the last of which is a 10-state variable (with values 0 through 9). If a prototype with an input vector $P.\boldsymbol{I}$="0015" is compared to the input vector $\boldsymbol{I}$="0111", the distance metrics defined above would be computed as shown in Figure 3.3.

Variable Name:	A	B	C	D	Out
Number of States:	2	2	2	10	2
Prototype:	**0**	**0**	**1**	**5** →	**1**
New input to be classified:	**0**	**1**	**1**	**1** →	**?**

Distance Metric	Distance
1. Hamming Distance:	0 + 1 + 0 + 1 = 2
2. Normalized Hamming Distance:	0 + 1/2 + 0 + 1/10 = 0.6
3. Absolute Difference:	0 + 1 + 0 + 4 = 5
4. Normalized Absolute Difference:	0 + 1/2 + 0 + 4/10 = 0.9
5. Euclidean Distance:	0 + 1 + 0 + 16 = 17

Figure 3.3. Results of various distance metrics.

Note that these distance values would not be compared to each other but to corresponding values computed for the rest of the prototypes in the set.

3.5. Voting

Voting functions are used to allow several prototypes to work together in deciding on an output. When the k nearest prototypes are being used (with k>1), voting allows the k nearest neighbors to all have influence on the output value. Even when k=1 (i.e., only the closest prototype is used), there often are several prototypes which are equally close to the new input, and voting can be used to break the tie.

Voting can be done in several different ways. Let *Min* be the set of "closest" prototypes to a given input vector. Note that when there are ties for the k^{th} closest prototype, *Min* may have more than k prototypes in it. Also, let M_z be the set of prototypes P in *Min* with an output of z. That is,

$$M_z = \{P\text{: } P \in Min \text{ and } P.Z=z\}.$$

The following four values are obtained easily:

- $p_z = |M_z|$, the number of prototypes in M_z.
- f_z = the sum of frequency values for all P in M_z, given by:

$$f_z = \sum_{P \in M_z} P.f$$

- c_z = the sum of conflict values for all P in M_z, given by:

$$c_z = \sum_{P \in M_z} P.c$$

- g_z = the "overall integrity" for all P in M_z, given by:

$$g_z = \frac{f_z}{f_z + c_z}$$

These values can be used in various ways to determine what the voting power will be for each output value z, as explained below.

3.5.1. *Prototype Voting.* One way to define the voting function is to allow each prototype to get one vote, regardless of its frequency. The voting function for Prototype Voting (PV) is:

$$\text{PV}(Min,z)=p_z \qquad \text{(Eq. 3.6)}$$

3.5.2. *Frequency Voting.* An alternative is to sum the frequency values of all the prototypes with the output z. This kind of voting gives more strength to prototypes which represent more instances in the training set. Frequency Voting (FV) is computed as:

$$\text{FV}(Min,z)=f_z \qquad \text{(Eq. 3.7)}$$

3.5.3. *Integrity Voting*. Another method is to use the overall integrity of the prototypes in M_z. This style of voting tends to give more strength to prototypes which are fairly undisputed, even if they do not represent as large a population in the original training set. The Integrity Voting (IV) function is:

$$\mathrm{IV}(Min,z)=g_z \qquad \text{(Eq. 3.8)}$$

3.5.4. *No Voting*. Of course, the simplest method of handling ties for the closest prototype is to ignore voting altogether. That is, arbitrarily pick one of the prototypes and use its output. For such a scheme it does not make sense to use a k>1 because only one prototype is being used, so it may as well be one that at least ties for being the closest one to the input.

3.6. First-order Features

One way to determine which variables are most important is to find those which tend to have a strong correlation with the output value. In order to discover critical variables, another kind of prototype called a *first-order feature* is created for each input variable of each instance. Each feature has only the output and one input variable asserted. The rest of the input variables are "don't care" variables ("*").

Combinations of variables can be used instead of only one variable at a time and thus create higher-order features, but such methods run into exponential factors in both storage and computation time and are thus not pursued in this research. Therefore the term *feature* is used to mean a first-order feature.

Example 3.2. During the learning phase, the instance "0015→1" would cause the four features listed in Figure 3.4 to be generated.

If any of these features already exist in the system, the new one is discarded, and the old one's frequency is incremented and its integrity recomputed. If any features existed which had the same input but a different output (e.g., 0 * * *→0), then both features would increment their conflict and recompute their integrity. The first few instances in the training set normally add quite a few features to the system, but later instances usually find matches for most of the features they add.

0 * * * -> 1	Frequency: 1	Conflict: 0	Integrity: 1.0
* 0 * * -> 1	Frequency: 1	Conflict: 0	Integrity: 1.0
* * 1 * -> 1	Frequency: 1	Conflict: 0	Integrity: 1.0
* * * 5 -> 1	Frequency: 1	Conflict: 0	Integrity: 1.0

Figure 3.4. First-order features created from the instance "0 0 1 5→1".

Example 3.3. In figure 3.5, a few instances from a training set are listed, along with the features that would be generated for the first and second variables. Note that the conflict field is simply the sum of the frequencies of all other prototypes with the same inputs.

Instances: A B C D	Instances: Out	First-order Features: A B C D	First-order Features: Out	Frequency	Conflict	Integrity
0 1 1 3	0	0 * * *	0	1	3	1/4=.25
0 0 0 0	1	0 * * *	1	3	1	3/4=.75
0 0 1 0	1	1 * * *	0	1	1	1/2=.50
0 1 1 5	1	1 * * *	1	1	1	1/2=.50
1 1 0 1	0	* 0 * *	1	3	0	3/3=1.0
1 0 1 0	1	* 1 * *	0	2	1	2/3=.66
⋮		* 1 * *	1	1	3	1/3=.33
		⋮				

Figure 3.5. Features generated from instances.

Prototypes and features are very similar things. In fact, a first-order feature is simply a prototype with the restriction that only one input value is asserted.

There are several ways to use first-order features for generalization. Given a list of features L

and an input vector $\boldsymbol{I}$ to be classified, let *Match* be defined as the set of all features that match $\boldsymbol{I}$. Furthermore, let $Match_i$ be the subset in *Match* of all the features which have variable i asserted. Finally, let $Best_i$ be the feature in $Match_i$ which has the highest frequency, i.e.,

- $Match_i = \{F\text{: } F \in L \text{ and } F.\boldsymbol{I}_i = \boldsymbol{I}_i \text{ and } F.\boldsymbol{I}_i \neq \text{“*”}\}$, for $1 \leq i \leq m$.

- $Match = \{F\text{: } F \in L \text{ and } F.\boldsymbol{I} \text{ matches } \boldsymbol{I}\} = \bigcup_{i=1}^{m} Match_i$

- $Best_i$ = the $F \in Match_i$ with maximum value for $F.f$, for $1 \leq i \leq m$.

For each i, all the features in $Match_i$ have variable i asserted to the same value, because they all match $\boldsymbol{I}_i$. Also, all features in $Match_i$ for each i have different output values, or they would have been combined into the same feature during learning.

Match

A	B	C	D		Out	Frequency	Conflict	Integrity
0	*	*	*	->	0	1	3	.25
0	*****	*****	*****	**->**	**2**	**3**	**1**	**.75**
*****	**1**	*****	*****	**->**	**0**	**6**	**4**	**.60**
*	1	*	*	->	2	4	6	.40
*	*	2	*	->	0	2	4	.33
*****	*****	**2**	*****	**->**	**2**	**4**	**2**	**.67**
*****	*****	*****	**3**	**->**	**1**	**4**	**1**	**.80**
*	*	*	3	->	2	1	4	.20

Best

A	B	C	D		Out	Frequency	Conflict	Integrity
0	*	*	*	->	2	3	1	.75
*	1	*	*	->	0	6	4	.60
*	*	2	*	->	2	4	2	.67
*	*	*	3	->	1	4	1	.80

Figure 3.6. *Match* and *Best* for the input “0 1 2 3”.

Example 3.4. Suppose that the input $\boldsymbol{I}$=“0 1 2 3” was received by the system. Figure 3.6 shows *Match*, a list of all the features which might match $\boldsymbol{I}$. Those features shown in bold have the highest frequency for the variable they assert, and are selected to be in the set *Best*, as shown on the right.

Following are six ways of using first-order features to generalize.

3.6.1. *Highest Frequency*. One way to use first-order features in generalization is to select the single feature from *Best* that has the highest frequency. This feature occurred more often than any other matching feature, so its output might be likely to be correct.

3.6.2. *Highest Integrity*. A similar method is to select the single feature in *Best* that has the highest integrity, and use its output. This feature is relatively undisputed, although it might have occurred less frequently than another feature.

3.6.3. *Voting by Best*. Another method is to allow the m features in *Best* to vote for their asserted outputs, using Frequency Voting (see Equation 3.7).

3.6.4. *Integrity Voting by Best*. Similarly, the m features in *Best* can use Integrity Voting (Equation 3.8), and the output value which receives the highest voting value from the voting function IV(*Best*) is used as the system’s output.

3.6.5. *Voting by All Matches*. Instead of limiting the voting to one or m best features, all the features in *Match* can vote using Frequency Voting.

3.6.6. *Integrity Voting by All Matches*. Finally, all the features in *Match* can vote using Integrity Voting.

Match

	A	B	C	D		Out	Frequency	Conflict	Integrity
M_1.	0	*	*	*	->	0	1	3	.25
M_2.	**0**	*****	*****	*****	**->**	**2**	**3**	**1**	**.75**
M_3.	*****	**1**	*****	*****	**->**	**0**	**6**	**4**	**.60**
M_4.	*	1	*	*	->	2	4	6	.40
M_5.	*	*	2	*	->	0	2	4	.33
M_6.	*****	*****	**2**	*****	**->**	**2**	**4**	**2**	**.67**
M_7.	*****	*****	*****	**3**	**->**	**1**	**4**	**1**	**.80**
M_8.	*	*	*	3	->	2	1	4	.20

Best

	A	B	C	D		Out	Frequency	Conflict	Integrity
B_1.	0	*	*	*	->	2	3	1	.75
B_2.	*	1	*	*	->	0	6	4	.60
B_3.	*	*	2	*	->	2	4	2	.67
B_4.	*	*	*	3	->	1	4	1	.80

Figure 3.7. Features matching input “0 1 2 3”.

Example 3.5. Given an input vector ***I***=“0 1 2 3”, the set of matching features M_1-M_8 is shown on the left-hand side of Figure 3.7 above. Those with the highest frequency for each variable are selected to be in the *Best* set, B_1-B_4, shown on the right. Each of the six methods above can be used to generalize in this case, with varying results:

1. *Highest Frequency*: B_2 (which is the same as M_3) is selected, because it has the highest frequency of all the features in *Best*, resulting in an output value of 0. Note that since *Best* is made up of the features in *Match* with the highest frequency, this feature also has the highest frequency of all the features in *Match* as well.

2. *Highest Integrity*: Feature B_4 (or M_8) is selected because it has the maximum integrity of any of the features in *Best* (or *Match*), resulting in an output value of 1.

3. *Voting by Best*. When features B_1 through B_4 vote for their outputs, an output value of 0 gets 6 votes from B_2, an output of 1 gets 4 votes from B_4, and an output of 2 gets 3+4=7 votes from B_1 and B_3. Thus, FV(*Best*) chooses an output of 2. Using the voting function notation, the following table summarizes the voting by the features in *Best*:

z	f_z	c_z	g_z
0	6	4	.60
1	4	1	.80
2	7	3	.70

Table 3.8. Results of voting by *Best* in Figure 3.7.

4. *Integrity Voting by Best*. Using the same table, an output of 1 has the highest overall integrity (.80), even though its frequency (4) is not the greatest.

5. *Voting by All Matches*. Looking again at Figure 3.7, an output of 0 gets 1+6+2=9 votes, an output of 1 gets 4 votes, and an output of 2 gets 3+4+4+1=12 votes, so FV(*Match*) chooses an output of 2. Again using the voting function notation, Table 3.9 is obtained from *Match*:

z	f_z	c_z	g_z
0	9	11	.45
1	4	1	.80
2	12	13	.48

Table 3.9. Results of voting by *Match* in Figure 3.7.

6. *Integrity Voting by All Matches*. Using Table 3.9, an output of 1 (i.e., z=1) again has the highest overall integrity (g_1=.80) even though it occurs less often than the output value of 2.

3.7. Confidence Levels

In the preceding discussion, the use of distance metrics and first-order features have been considered separately, and each has been enhanced by some method of voting in order to resolve conflicts when multiple prototypes have contended for differing output values.

In this section, methods are discussed for combining the use of distance metrics and first-order features in various ways. The *confidence level* of prototypes and features are computed in several ways, as explained below, and they are used to influence distance calculations as well as voting power. The generalization styles using confidence levels use the same overall algorithm (Algorithm 3.1) as the distance metric styles, except that the functions VOTE and DIST are modified. More alternatives for the definitions of the functions VOTE and DIST in Algorithm 3.1 are presented in Sections 3.8 and 3.9, respectively.

3.8. Using Confidence Levels in Voting

When distance metrics are used by themselves, prototypes which are equally "close" to the new inputs being classified are given a number of votes equal to their frequency. That is, they are given one vote for each of the instances in the training set that they represent. This voting power does not, however, take into account how "good" or confident the prototypes are about their guesses.

It would be advantageous at times to allow prototypes with a high *confidence level* (CL) to have more influence over the final output. This confidence level can be based on the prototype's integrity, or on the integrity of its features.

It is important to note that the term *confidence* usually denotes an estimate of the probability of a correct output. The term *confidence level*, as used in this thesis, is a relative value, and does not necessarily give the precise probability of a correct answer, although a higher confidence level usually indicates a higher probability of being correct.

Several ways of using prototypes and features to calculate confidence levels are given below. In each case the voting power of a prototype is equal to its frequency multiplied by the confidence level. That is, VOTE(L, z), the number of votes cast for output z by the prototypes in list L is given by:

$$\mathrm{VOTE}(L,z) = \sum_{P \in Lz} P.f * \mathrm{CL}(P)$$

where L_Z={P: $P \in L$ and $P.Z$=z} and CL(P) is the confidence level of prototype P. Six possible definitions for the function CL are given below.

3.8.1. *CL=Prototype's Integrity*. The first method is to use the prototype's integrity as the confidence level, since prototypes which often are contradicted should not be trusted. For example, if a prototype "011 $\rightarrow$ 1" has an integrity of .30, then 70% of the time the input of "011" appeared as an input vector in the training set, the output was something other than 1. Formally, the Prototype's Integrity function is given as simply:

$$\mathrm{PI}(P) = P.g = \frac{P.f}{P.f + P.c} \qquad \text{(Eq. 3.9)}$$

That is, if PI is used as the definition of CL, then CL(P)=PI(P)=$P.g$.

Example 3.6. Figure 3.10, shown below, lists several prototypes and shows their input values (for input variables A, B, and C), output value (for output variable Z), frequency, conflict, and integrity. Again the conflict is equal to the sum of the frequency values for all prototypes with the same input value but different output values.

ABC -> Z	Frequency	Conflict	Integrity
1 0 1 -> 0	40	60	.40
1 0 1 -> 1	31	69	.31
1 0 1 -> 3	29	71	.29
1 1 0 -> 0	30	90	.25
1 1 0 -> 1	40	80	.33
1 1 0 -> 2	50	70	.42

Figure 3.10. A list of several prototypes.

In this example, suppose that the input vector "111" is used, and is found to be equally close to all of the prototypes listed. Using straight voting, the results are as follows:

- Votes for output of 0: 40 + 30 = 70
- Votes for output of 1: 31 + 40 = 71
- Votes for output of 2: 50 = 50
- Votes for output of 3: 29 = 29

and the output is 1. However, weighting each prototype's voting power by its confidence level yields:

- Weighted votes for output of 0: $40 * .40 + 30 * .25 =$ 23.50
- Weighted votes for output of 1: $31 * .31 + 40 * .33 =$ 22.94
- Weighted votes for output of 2: $50 * .42$ = 21.00
- Weighted votes for output of 3: $29 * .29$ = 8.41

and in this case the output 0 wins by a small margin.

In practice, most instances in a training set are not directly contradicted. That is, it is rare to have two instances with the exact same input vector and different outputs. Therefore, the integrity of the overwhelming majority of prototypes will be 1.0, and weighting votes by this value has little effect (if any) on which output is selected.

First-order features, however, are contradicted quite often, because the values of input variables usually are not correlated completely with the output. Furthermore, the features matching the prototype can give us indirectly an indication of how "good" the prototype itself is.

For example, the prototype "100→1" is matched by the features "1**→1", "*0*→1", and "**0→1". In fact, the prototype created these features in the first place during the learning phase. These features are said to be those which are *associated* with the prototype.

That is, P is associated with one feature for each of its asserted input variables. For each input variable i, $1 \le i \le m$, P is associated with one feature, written $\mathrm{P.F}_i$ (not to be confused with $P.f$, the frequency of P). Furthermore, the feature $\mathrm{P.F}_i$ has the same output value and same asserted input value for variable i (i.e., $P.F_i.\boldsymbol{I}_i = P.\boldsymbol{I}_i$ *and* $P.F_i.Z = P.Z$).

If the features associated with a prototype have a high integrity, then the prototype itself might be considered good. The remainder of this list contains ways of using first-order features to compute the confidence level of a prototype.

3.8.2. *CL=Prototype's Best Feature's Integrity*. One simple way to use features to compute a prototype's confidence level is to find the feature associated with the prototype that has the highest integrity and to use that integrity as the prototype's confidence level. This integrity gives the probability that the output is correct given the single variable's asserted value. For example, if the prototype "100→1" had a feature "1**→1" with integrity .95 and features "*0*→1" and "**0→1" with integrity .7, .95 can be used as the confidence level for "100→1" since the output is "1" 95% of the time that the first input variable is "1", and in "100→1" the first input variable is indeed "1."

The Best Feature Integrity function is given as:

$$\mathrm{BF}(P)=\max\{P.F_i.\mathrm{g},\ 1 \le i \le m\} \qquad \text{(Eq. 3.10)}$$

3.8.3. *CL=Average Integrity of the Prototype's Features*. An alternative approach is to average the integrity values of all of a prototype's features to get a confidence level. In the above example, the confidence level would be (.95 + .70 + .70) / 3 = .78. The rationale behind this measure is that although the first feature tells us that the output is "1" 95% of the time when the first variable is "1", the second and third feature tell us that 30% of the time that the second and third variable are zero (respectively), the output is something other than 1. Since these facts disagree, one way to resolve the matter is to average the integrity values together.

The Average Feature Integrity function is given as:

$$\mathrm{AF}(P) = \frac{\sum_{i=1}^{m} P.F_i.g}{m} \qquad \text{(Eq. 3.11)}$$

3.8.4. *CL=Overall Integrity of the Prototype's Features*. An alternate way to combine the integrity values of the features is to sum their frequencies and conflicts, and take an overall integrity for them. The Overall Feature Integrity function is given by:

$$\mathrm{OF}(P) = \frac{\sum_{i=1}^{m} P.F_i.f}{\sum_{i=1}^{m} (P.F_i.f + P.F_i.c)} \qquad \text{(Eq. 3.12)}$$

One possible drawback with this scheme is that it has the effect of giving features which occurred more often more weight. This may not be desirable because it gives more weight to variables with a low cardinality, and there is little reason to do so.

For example, out of 1000 instances with Boolean output, suppose a feature with a Boolean input asserted has an average frequency of 250, while a 10-state variable has an average frequency of 50. The Boolean variable's feature is not any "better", it just covers a larger proportion of the input space. That is, it has one-fifth the number of possible values, so it will on average have five times the frequency. Thus, in Equation 3.12, the f and c values will on average be five times larger for Boolean inputs than for 10-state inputs. This will give correspondingly more weight to the Boolean input variables.

3.8.5. *CL=Weighted Overall Integrity of Prototype's Features*. In order to "normalize" the overall integrity computed above, the cardinality of each variable is used to weight the frequency and conflict before the integrity is computed. This is shown below in the definition of the Weighted Overall Feature Integrity function:

$$\mathrm{WOF}(P) = \frac{\sum_{i=1}^{m} |A_i| * P.F_i.f}{\sum_{i=1}^{m} |A_i| * (P.F_i.f + P.F_i.c)} \qquad \text{(Eq. 3.13)}$$

where $|A_i|$ is the cardinality of the i^{th} input variable.

In the above example, the Boolean feature would have a new weighted frequency of 2 x 250 = 500, and the 10-state feature would get 10 x 50 = 500, evening things out.

3.8.6. *CL=Weighted Average Integrity of the Prototype's Features*. The average integrity of the prototype's features could also be weighted by the number of states for each variable, as follows:

$$\mathrm{WAF}(P) = \frac{\sum_{i=1}^{m} |A_i| * P.F_i.g}{\sum_{i=1}^{m} |A_i|} \qquad \text{(Eq. 3.14)}$$

This method has the same drawback as OF (Eq. 3.12), above, except that it effectively "unnormalizes" the average taken in AF (Eq. 3.11). As expected, OF and WAF never did better than their normalized counterparts—WOF and AF, respectively—on real-world applications.

	A B C -> Out	Frequency	Conflict	Integrity
Prototype:	0 1 2 -> 0	4	1	.80
First-order Features:	0 * * -> 0	500	200	.71
	* 1 * -> 0	200	100	.66
	* * 2 -> 0	100	0	1.0
Number of States:	2 5 10 2			

Figure 3.11. The prototype "0 1 2 $\rightarrow$ 0" and its corresponding features.

Example 3.7. A prototype ("012$\rightarrow$0") is shown in Figure 3.11, along with its corresponding features. Given that the cardinalities of the three variables are 2, 5, and 10, respectively, the six different kinds of confidence levels would be computed as shown in Table 3.12.

	Description of CL Function	Name	Computation	Result
1.	CL(P)=Prototype's integrity:	PI(P)	$= P.g$	= .80
2.	CL(P)=Best feature's integrity:	BF(P)	= max{.71, .66, 1.0}	= 1.0
3.	CL(P)=Average feature integrity:	AF(P)	$= \frac{.71+.66+1.0}{3}$	= .79
4.	CL(P)=Overall feature integrity:	OF(P)	$= \frac{500+200+100}{700+300+100}$	= .72
5.	CL(P)=Weighted overall integrity:	WOF(P)	$= \frac{500*2+200*5+100*10}{700*2+300*5+100*10}$	= .77
6.	CL(P)=Weighted average integrity:	WAF(P)	$= \frac{.71*2+.66*5+1.0*10}{2+5+10}$	= .87

Table 3.12. Six different confidence levels for the prototype in Figure 3.9.

Note that each of these values for the confidence level would be used by a different generalization style.

3.9. Using Confidence Levels to Weight Distance Metrics. In Section 3.8 confidence levels are used to weight the voting power of prototypes. This section shows how they can be used to influence the calculation of the various distance metrics before voting even takes place. The idea is that if a variable which is a good predictor of the output is mismatched, then the "distance" should be considered greater than a mismatch on a variable which has little correlation with the output.

This can be done in several ways. In each of the methods presented below, the distance for each variable is weighted by some confidence level before being summed with the distances of the other variables. In terms of Algorithm 3.1, this section introduces weighting schemes which can be used to modify each of the five distance metrics presented in Section 3.4, and thus adds several more possible definitions for the DIST function.

For example, if CL(P,i) is the weight placed on input i of prototype P, then the Absoulte Difference metric would be altered as follows:

$$\mathrm{AD}(\boldsymbol{A},P) = \sum_{i=1}^{m} |\boldsymbol{A}_i - P.\boldsymbol{I}_i| * \mathrm{CL}(P,i),$$

3.9.1. *Using an Overall Variable Correlation.* One intuitive measure for how much weight to give each variable is the overall correlation which that variable has with the output. If a variable has a high correlation with the output, then it should be considered more important and weighted heavier when determining the distance.

For example, consider the input "111" and two instances "110$\rightarrow$1" and "101$\rightarrow$0". Assume also that the correlation for the three input variables are .2, .5, and .1, respectively. Then the second variable is usually a much better predictor of the output than the first or third variable. Since the second variable is mismatched in "101$\rightarrow$0", its distance is 0+.5+0 = 0.5. The third variable is mismatched in "110$\rightarrow$1", and its distance is 0+0+.2 = 0.2. The second instance is considered "closer" because the variable it was mismatched on was less critical.

The correlation for each variable can be found in the following manner. For each variable i, consider only the set of features L_i with variable i asserted. Then for each unique input value of that variable, pick the feature with the highest frequency (or, equivalently, the highest integrity), resulting in the set S_i, which can be defined as:

$S_i=\{F\text{: } F\in L_i,\ F_i \text{ asserted, } F.f>G.f \text{ for any } G\in L_i \text{ with } G.I_i=F.I_i\}$

The correlation for variable i can then be found by:

$$\mathrm{COR}(i)=\frac{\sum_{F\in Si} F.f}{\sum_{F\in Si}(F.f+F.c)} \qquad \text{(Eq. 3.15)}$$

When the sums of the frequencies and conflicts (i.e., the denominator of Eq. 3.15) equal the number of instances, this measure is the actual correlation of that variable with the output. If, on the other hand, there are “don’t know” values for that variable in some instances, the measure only tells the correlation between that variable and the output *when that variable is asserted.* Section 3.10 discusses this point in more detail.

Example 3.7. Given the three features in Figure 3.13 below, the first prototype is the only one with “1” asserted for the first input, and the second one, “2**→1” has the highest frequency of the two with “2” asserted for the first input, so the first and second features comprise S_1, and are used to compute the correlation for the first variable.

Input	->	Out	Frequency	Conflict	Integrity
1 * *		0	1	0	1.0
2 * *		1	110	90	.55
2 * *		2	90	110	.45

Figure 3.13. Three feature prototypes with the first input asserted.

Using Equation 3.15,

$$\mathrm{COR}(i)=\frac{\sum_{F\in Si} F.f}{\sum_{F\in Si}(F.f+F.c)}=\frac{1+110}{(1+0)+(110+90)}=\frac{111}{201}=.55$$

Thus, the first variable has a 55% correlation with the output.

To see how the correlation might be used during generalization, suppose that the second and third variables were found to have a 35% and 80% correlation with the output, respectively. Figure 3.14 below shows three prototypes and the input vector “2 1 0”.

Input Vector
2 1 0

	Prototypes	Hamming Distance	Weighted Hamming Distance
1.	0 1 0->0	1+0+0 = 1	.55 + 0 + 0 = .55
2.	2 0 0->1	0+1+0 = 1	0 + .35 + 0 = .35
3.	2 1 1->2	0+1+1 = 2	0 + .35 + .80 = 1.15

i	COR(i)
1	.55
2	.35
3	.80

Figure 3.14. Using Correlation to weight Hamming Distance

Note that the Hamming Distance between the input vector and the first two prototypes is 1. However, when the individual variables’ distances are weighted by COR(i), the second prototype is considered to be closer to the input vector, because the variable it mismatched on had a lower overall correlation with the output (and thus a lower confidence level).

Some alternatives that did not measure up. Several other measures for an overall correlation were considered, but each proved to be less satisfactory than that given above. For example, one alternative is to pick the features for each input variable that have the highest integrity for each unique input value (as was done in the above scheme), and then average their integrity values to get the weight for that variable. This is similar to the approach above, but does not take into account the actual frequency and conflict being used to produce the integrity values. For example, given the three features above, the best integrity values are 1.0 and .55 for the two different input values. The average of the best integrity values is .775, but this is misleading, because in 200 out of 201 cases, the integrity is either .55 or .45, but the single remaining case brings the average integrity for the entire variable up to .775.

Another discarded alternative is to average the integrity values of all of the features with the i^{th} input variable asserted. However, it turns out that this average equals the number of different input values for that variable divided by the number of features with that variable asserted. This average is independent of the actual integrity values themselves, because the integrity values for each unique input value always adds up to 1.0.

For example, given the three features "1**→0", "1**→1", "2**→3" with integrity values of .99 .01, and 1.0, respectively, the average integrity is (.99+.01+1.0)/3 = 2/3. This is simply because the first variable has two different input values, and there are three features altogether.

Another alternative is to take the one feature for each input variable which has the highest integrity and use that as the weight for the variable. However, there is quite often at least one feature for each input variable which has an integrity of 1.0, as occurs when an input value appears only once and is thus never contradicted. Therefore, the weight for each variable would likely be 1.0 for almost all variables, which means that the weight would not have any effect.

3.9.2. *Use the Integrity of the Prototype's i^{th} Feature*. The integrity values of the individual features associated with a prototype also can be used to weight the distance for each variable. Given a prototype *P*, the distance for each variable (using any distance metric) is weighted by the integrity of *P*'s i^{th} feature, i.e.,

$$\mathrm{PF}(P,i)=P.F_i.g \qquad \text{(Eq. 3.16)}$$

If a prototype is mismatched on a variable for which its feature has a high integrity, the distance—or error—is weighted heavier than if the feature had a low integrity. In this way, input variables of a prototype which do not seem too confident anyway are not considered as erroneous when they are mismatched.

Instance ***I***: 1 1 0			
Prototype P1:	0 0 0 -> 0	Integrity	Hamming Distance
$P1.F_1$:	0 * * -> 0	.75	
$P1.F_2$:	* 0 * -> 0	.22	HD(***I***, *P1.**I***) = 1 + 1 + 0 = 2
$P1.F_3$:	* * 0 -> 0	.43	HD(**I**, *P2.**I***) = 0 + 1 + 1 = 2
Prototype P2:	0 1 2 -> 1	Integrity	Hamming Distance weighted by Feature Integrity
$P2.F_1$:	0 * * -> 1	.55	
$P2.F_2$:	* 1 * -> 1	.98	HDwF(***I***, *P1.**I***) = .75 + .22 + 0 = .97
$P2.F_3$:	* * 2 -> 1	.27	HDwF(***I***, *P2.**I***) = 0 + .98 + .27 = 1.25

Figure 3.15. Weighting Hamming Distance by PF(*P*).

Example 3.8. Figure 3.15 shows an instance *I*, and two prototypes, P1 and P2. It also shows the features associated with each prototype, along with the integrity of each feature. In the upper-right portion of the figure, the Hamming Distance is computed between *I* and each prototype. Note that the distance is the same (i.e., 2) in each case. Weighting the distance by the prototype's corresponding features, however, results in a smaller distance for P1, because it was not very confident of the value for the variable it mismatched on.

This scheme can backfire, however, when a prototype is made up entirely of "poor" features, because then it is considered “close” to almost all input vectors.

For example, consider a bad configuration in which one “noisy” prototype is associated with four features whose integrity values are all .01. Then even if it is mismatched on all 4 inputs the distance would be only .04, so unless another prototype exactly matched the input, this noisy prototype would almost always be the “closest.”

3.9.3. *Weighting the Overall Distance*. One could compute the overall distance without weighting the variables, and then weight the final distance using a confidence level for the prototype, i.e.,

$$\mathrm{WDIST}(\boldsymbol{I},P.\boldsymbol{I}) = \mathrm{CL}(P) * \mathrm{DIST}(\boldsymbol{I},P.\boldsymbol{I})$$

Any of the confidence levels discussed in Section 3.8 could be used for the function CL. Unfortunately, this scheme often has the effect of making “good” prototypes less likely to be chosen, since higher numbers are considered “worse” where distance is concerned. Using the reciprocal ($1/\mathrm{CL}(P)$) or inverse (1.0-CL(P)) would likely have better success.

As expected, this scheme did not do well empirically in general. However, there was at least one application (“Lung Cancer” database) on which weighting the overall distance by the integrity of the best feature of a prototype (i.e., BF(P)) did better than any of the other styles used, so it may have potential in some cases.

3.10. Using Confidence Levels with First-order Features. Section 3.9 presented a measure of correlation for variable i, COR(i). In Section 3.6, a list $Best_i$ was found containing the best feature for each of the i input values in the input vector I. This section combines these two ideas into a method called *Highest Correlation* by using the output of the feature whose asserted input variable has the highest overall correlation with the output. This is done in the following way.

Given COR(i) and $Best_i$ for $1 \le i \le m$, j is chosen such that COR(j) is greatest. That is,

$$j\text{: } \mathrm{COR}(j)=\max\{\mathrm{COR}(i),\ 1 \le i \le m\}$$

Then the output of $Best_j$ (i.e., $Best_j.Z$) is used as the output value.

For example, given the input “1 0 1”, assume that the features with the highest integrity for each variable are as follows:

$Best_1$=“1 * * → 1”, with integrity of .80
$Best_2$=“* 0 * → 0”, with integrity of .50
$Best_3$=“* * 1 → 0”, with integrity of .75

Also, let the overall correlation for the three input variables be .65, .42, and .85, respectively. Using Highest Integrity method from Section 3.6.2, the first feature would be chosen, and the output would be 1. Using the Highest Correlation method, however, the overall correlation values tell us that in general, the third variable is a better predictor of the output. Therefore, the third feature is chosen and the output is 0.

Somewhat surprisingly, the Highest Correlation scheme did much better empirically than the Highest Integrity scheme. This is surprising because the Highest Correlation method essentially throws away all the information contained in all of the inputs of the training set except for the one with the highest correlation. The problem with the other scheme, however, may be that there are usually so few of each feature that the sample size does not allow for an accurate estimate of the feature’s true integrity.

For an application with no unknown input values, this scheme should be about as accurate as the best input variable’s correlation with the output. Sometimes this is better than any of the other schemes, but it is weak in other cases because it ignores all but one input variable, and does not exploit any higher-order relationships between several input values and the output.

Using higher-order features to get a correlation between pairs or sets of input variables and the output has the potential of being more effective. However, doing so runs into combinatorial growth in terms of storage and processing time and exacerbates the problem of small sample sizes.

When applications do have unknown input values, the correlation as computed in the manner described above may be optimistic, since it only holds when the input variable is asserted. For example, in one simulation involving the "Audiology" database, one variable has a 75% correlation with the output. However, the value for that variable is unknown in all but 4 out of the 200 instances, and then is only correlated 3 out of 4 times. Thus, its correlation was really 3/200 = 1.5%, not 75%.

When there are no unknown input values, this method guarantees a generalization accuracy of at least max{COR(i), 1≤i≤m}. However, if there are unknown inputs the frequency and conflict do not add up to the number of instances, so only the following generalization accuracy is guaranteed:

$$\max\{\frac{\sum_{F \in S_i} F.f}{U_i + \sum_{F \in S_i} F.f + F.c}, 1 \le i \le m\} = \max\{\frac{\sum_{F \in S_i} F.f}{t}, 1 \le i \le m\}$$

where U_i is the number of unknown input values appearing in the i^{th} variable, t is the number of instances in the training set, and S_i is as defined in Section 3.9.1.

During execution, if the input value of the variable with the highest correlation is unknown, there will be no feature corresponding to it, so the variable with the next best correlation must be used instead. In other words, the feature corresponding to the variable with the highest correlation—of those which are asserted in the input vector—is used to predict the output. In this way, a degree of accuracy is obtained which lies somewhere between the guaranteed minimum and the optimistic measure.

3.11. Squashing Function

Sometimes it is advantageous to use some sort of "squashing" function to make decisions more definite. For example, when computing a confidence level to use in weighting a prototype's voting power, it may help to allow prototypes with a high confidence level to get more than a linear increase in voting power while those with a very low confidence level get less than even their small linear share of the votes.

In neural networks such as the backpropagation network, the sigmoid function is often used as a squashing function. This function, shown below, takes any real value and returns a value between 0 and 1.

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

As x approaches infinity, $sigmoid(x)$ approaches 1; as x goes to negative infinity, $sigmoid(x)$ approaches 0; and $sigmoid(0)=1/2$.

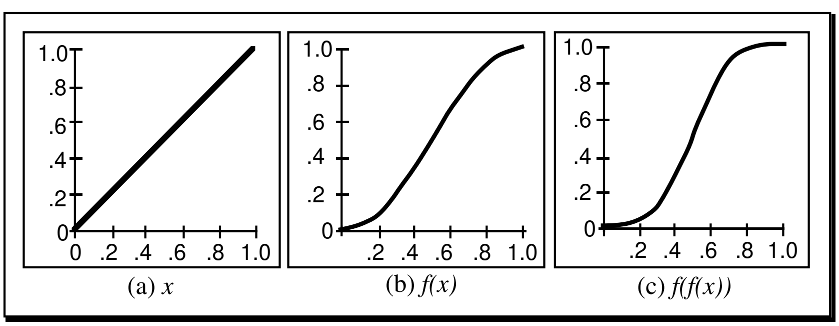

Figure 3.16. The effect of the squashing function f (Eq. 3.17)
(a) before squashing, (b) after one pass through f, and (c) twice through.

3.11.1. *Cosine-based Function.* All of the confidence levels discussed in this thesis are in the range of 0 to 1 already, so what is really needed is a function which moves values near 1.0 closer to 1.0 and values near zero closer to zero. The cosine function has the approximate shape needed, so if it is inverted, scaled down by half, and transposed upwards, the result is

$$f(x) = \frac{1-\cos(\pi * x)}{2} \qquad \text{(Eq. 3.17)}$$

with a corresponding graph as shown in Figure 3.16 (b) above.

The curvature is actually fairly gentle, however, so running the output through the same function (i.e., $f(f(x))$) again gives a more pronounced squashing effect, as shown in Figure 3.16 (c). The Double Cosine function therefore is defined as:

$$\text{DC}(x)=f(f(x))=\frac{1-\cos\left(\pi * \left(\frac{1-\cos(\pi * x)}{2}\right)\right)}{2} \qquad \text{(Eq. 3.18)}$$

3.11.2. *Squaring*. Another approach is simply to square the confidence level. This makes everything (except 1.0) smaller, but it reduces small numbers by more than it reduces larger ones. This would therefore be another way to reduce the power of weak prototypes.

3.12. Summary of Generalization Styles

To summarize, a list of the generalization styles presented in this thesis is given below:

I. Distance metrics (with or without frequency voting)
- A. Hamming Distance [HD($P.\boldsymbol{I},\boldsymbol{I}$)]
- B. Normalized Hamming Distance [NHD($P.\boldsymbol{I},\boldsymbol{I}$)]
- C. Absolute Difference [AD($P.\boldsymbol{I},\boldsymbol{I}$)]
- D. Normalized Absolute Difference [NAD($P.\boldsymbol{I},\boldsymbol{I}$)]
- E. Euclidean Distance [ED($P.\boldsymbol{I},\boldsymbol{I}$)]

II. First-order features
- A. Highest Integrity
- B. Highest Frequency
- C. Voting by Best
- D. Integrity Voting by Best
- E. Voting by All Matches
- F. Integrity Voting by All Matches
- G. Highest Correlation

III. Confidence levels
- A. For each distance metric, weight the prototypes' votes by CL(P)=
  - 1) Prototype Integrity [PI(P)]
  - 2) Best Feature Integrity [BF(P)]
  - 3) Average Feature Integrity [AF(P)]
  - 4) Weighted Overall Feature Integrity [WOF(P)]
- B. Use a squashing function on confidence levels A1-A4
  - 1) Double Cosine [DC(CL)]
  - 2) Square [CL^2]
- C. Multiply the distance of each variable i by DCL(P,i)=:
  - 1) The overall correlation for variable i [COR(i)]
  - 2) The integrity of the prototype's i^{th} feature [$P.F_i.g$]
  - 3) The prototype's best feature integrity [BF(P)]

Chapter 4
Implementation Considerations

Each of the generalization styles listed in the summary above was implemented and tested on several applications. Section 4.1 presents the learning algorithm that was implemented on a conventional (serial) computer. Section 4.2 discusses how learning and execution would work on a binary tree parallel architecture.

4.1. Learning Algorithm (Conventional)

4.1.1. Explanation of Learning Algorithm. Algorithm 4.1 below gives a pseudo-code implementation of the learning algorithm used for simulations on a serial machine.

Algorithm 4.1. Serial Learning Algorithm

```
Learn(instance set T);
    Let prototypeTree = BuildTree(T);
    ComputeConflict&Integrity(prototypeTree);
    Let featureList  = ExtractFeatures(T);
    Let featureTree = BuildTree(featureList);
    ComputeConflict&Integrity(featureTree);

BuildTree(instance set S): prototype;
    Inserts a set S of instances into a binary tree of prototypes,
        and returns the root.
    root = NULL;
    for each instance E in S
        root=InsertNode(root,E);
    return root;

InsertNode(prototypeNode N, instance E): prototypeNode;
    Inserts the instance E into the sub-tree N in such a way that N will
        be sorted first by input values, then by output value when needed.
    if N is NULL
        N=NewNode();   Create a new prototype.
        N.I=E.I;       Copy the input vector and output value from E.
        N.Z=E.Z;
        N.f=1;         Set the frequency to 1.
    else if (N.I>E.I) or (N.I=E.I and N.Z>E.Z)
        N.leftChild = InsertNode(N.leftChild,E);
    else if (N.I<E.I) or (N.I=E.I and N.Z<E.Z)
        N.rightChild = InsertNode(N.rightChild,E);
    else N.f = N.f+1;      E matches N, so increment the frequency.
    return N;   Returns the original N unless it was NULL,
                 in which case the new node N is returned.

ComputeConflict&Integrity(prototypeNode tree)
    Given a sorted binary tree of prototypes, computes the
        conflict and integrity for each prototype.
    Let P_1..P_n be the prototypes found from an inorder traversal of tree.
    Let pos = 1;
    while (pos < n)
        if (P_pos.I = P_pos+1.I)  Then there is a conflict, so identify the
            let last = pos + 1;          prototypes P_pos..P_last which all have the
            let sum = P_pos.f;       same input vector and sum their frequencies.
            while (P_pos.I = P_last.I)
                sum = sum + P_last.f;
                last = last + 1;
            for each prototype P_pos..P_last
                P.c = sum - P.f;           Compute the conflict.
                P.g = P.f / (P.f+P.c);     Compute the integrity.
            pos = last + 1;
        else
```

```
46.             P.c = 0;     The conflict is 0
47.             P.g = 1.0;  ...so the integrity is 1.0.
48.             pos = pos + 1;
49.       return;

50.   ExtractFeatures(instance set T):instance set;
51.       Returns a set containing one first-order feature for each asserted
52.           input variable in each instance in T.
53.       Let List be a list of features which is initially empty.
54.       for each instance E in T
55.           for each input variable i=1..m
56.               if (E.I_i  is  asserted)
57.                   F = NewNode();
58.                   F.I_1..F.I_m = '*';  Set all inputs to "don't care"
59.                   F.I_i = E.I_i;       Set input i to asserted value
60.                   F.Z = E.Z;           Set output to asserted value
61.                   add F to List;
62.       return List;
```

Given a training set T, learning takes place as follows. In line 2 of Algorithm 4.1, each instance of the training set is inserted into a binary tree of prototypes (the *prototype tree*) sorted by input and output values. If a prototype which exactly matches the instance already exists in the tree, then the matching prototype's frequency is incremented. Otherwise, a new prototype is created with the inputs and outputs of the new instance, and the frequency is set to 1.

Since the tree is sorted by input values first, all prototypes with the same input vector are contiguous in an inorder traversal of the tree. In line 3, the conflict field of each prototype is set to the sum of the frequency fields of all of the other prototypes with the same inputs (but different outputs). Once the conflict field of each prototype is found, the integrity can be computed for each prototype using the formula:

$$integrity = \frac{frequency}{frequency + conflict} \qquad \text{(Eq. 4.1)}$$

In line 4, each instance is broken up into m first-order features—one for each of the m input variables in an instance. These feature prototypes are inserted into another tree, the *feature tree*, and the frequency, conflict and integrity of these feature prototypes are computed in the same way as in lines 2 and 3.

4.1.2. Testing Generalization Accuracy. Once the system has created the prototype tree and feature tree, it is ready to generalize using any or all of the generalization styles described in this thesis. However, all of the styles do not always agree on what the output should be. In order to determine which style to use, a *test set* can be used to measure the generalization accuracy of each style. The style that is most accurate during subsequent generalization then can be chosen as the best style for that application. A *test set* is a collection of instances, just like a training set, but usually contains instances that are not part of the training set. The system uses only the inputs to predict what the output should be, and then compares its predicted output with the "real" output of the instance to see if it generalized correctly.

Usually a portion of the available examples are chosen to use as the training set during learning and then use the remaining examples as the test set to test generalization. This does not give an optimal representation of the generalization ability of the learning system, though, for two reasons. First, the learning system only gets to use part of the examples to learn from, which means there may be more information in the examples in the test set which it could have used to do better generalization. Second, a small test set would be a less dependable measure of the generalization accuracy than a larger test set, because it provides a smaller statistical sample.

In this system the measure of generalization accuracy is very important, because it determines which style of generalization to use during later execution. Therefore, *N-fold (leave-one-out) cross-validation* is used, so that out of n instances in each database, *n-1* instances can be used in the training set, and n instances can be used in the test set. This is accomplished as follows.

Since the prototype styles of generalization do learning in one pass of the data it is possible to learn the entire database and then temporarily "unlearn" each instance, one at a time. At that point it is as though the system had never seen the "unlearned" instance, so it can use the remaining prototypes to predict the output of the removed instance using each of the various generalization styles. Then each of the predicted output values is compared against the real output of the instance to determine whether each style was correct. Finally, the instance is relearned, the next one is unlearned, and the process is repeated *n* times.

4.2. Parallel Learning Algorithm

Prototype styles of generalization can be easily implemented on a parallel architecture to speed up execution time during generalization. Consider an architecture containing a binary tree of nodes such as that shown in Figure 4.1. Each node needs enough memory to store one example E (input vector $\boldsymbol{I}$ and output Z) and one prototype P (integers $\boldsymbol{I}_1..\boldsymbol{I}_m$, Z, f and c; and a real value g). When a prototype has not yet been stored in a node, that node and its prototype are both said to be *empty*. Each node also has a *parent*, a *left child*, and a *right child*. The parent of the root node is the *environment*.

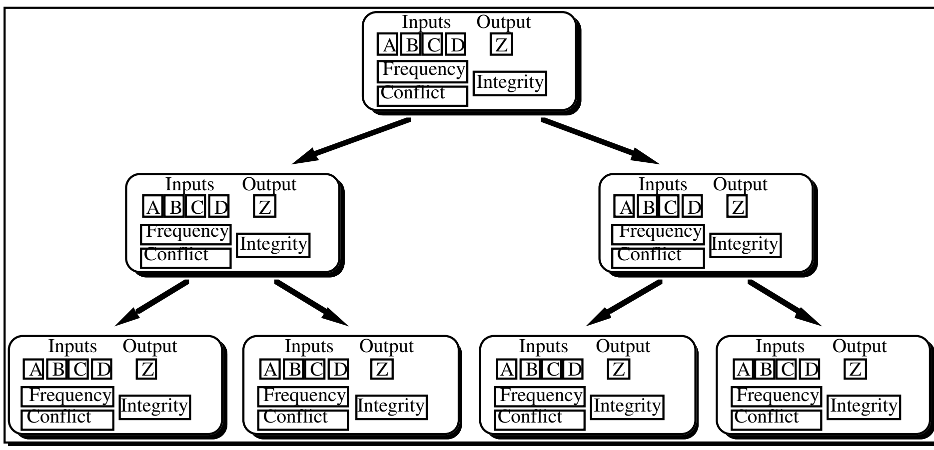

Figure 4.1. Binary tree architecture.

4.2.1. Explanation of the Learning Algorithm. Given a set of training instances T, and the set of nodes N in the tree, learning takes place as shown in Algorithm 4.2. The Environment routine is executed by an external system and presents the training set to the root of the binary tree. The PrototypeLearn routine is run by all the prototypes in parallel each time a new instance is to be learned.

Algorithm 4.2. Parallel learning algorithm.

Environment(training set T)
    for each instance E in T
        Broadcast E to *root*
        execute PrototypeLearn() in parallel for all prototypes
        Receive *match* flag and *conflict* from *root*
        if *match* flag is set to false
            Send E to the next free prototype node P;
            Let $P.c$ = *conflict;*
            Let $P.f = 1$;
            Let $P.g = P.f \,/\, (P.f + P.c)$;

PrototypeLearn()
    This routine is executed by all prototypes in parallel once for
        each instance learned.
    Receive broadcasted instance E from parent
    Broadcast E to left and right children (if non-empty)
    if ($P.\boldsymbol{I}$=$E.\boldsymbol{I}$ and $P.Z$=$E.Z$)   *If there is a match...*

```
    increment P.f by 1;              Increment frequency.
    let P.g=P.f/(P.f+P.c);           Recompute integrity.
    set match=true;
else set match=false;
if (P.I=E.I and P.Z≠E.Z)             If there is a conflict...
    increment P.c by 1;              Increment conflict.
    let P.g=P.f/(P.f+P.c);           Recompute integrity.
    let conflict=P.f;
else let conflict=0;                 E does not conflict with P.
Receive match flag and conflict value from any non-empty children;
if match=true or either child's match flag is true
    Send 'match=true' to parent;
else Send 'match=false' to parent;
Send sum of children's conflict with P's conflict to parent;
```

Each instance E in T requires a broadcast and gather. During the broadcast stage, E is passed down the tree to all the prototypes. Each prototype P in the tree checks to see if it is equivalent to E. If so, then P increments its frequency and recalculates its integrity. During the gather stage, E is not made into a new prototype if any prototype reports that it matched E.

If P conflicts with E, then it increments its conflict and recalculates its integrity. It also passes its frequency up the tree to be summed with any other conflicts so that if E is made into a new prototype, its *conflict* value is set initially to the correct value.

If, during the gather stage, no prototype reports that it has matched the example, then E is added in a new prototype at the next available free node in the tree. In this way, the tree is kept balanced at all times. Furthermore, the location of each prototype is independent of its contents, so any parallel architecture with some sort of fast broadcast-and-gather capabilities could be used for prototype generalization.

The conflict and integrity are kept up to date throughout the learning process, so learning is incremental. This is one advantage over the serial learning algorithm.

The first-order features can be extracted from the training set T using ExtractFeatures from Algorithm 4.1, and a feature tree can be created using the routines in Algorithm 4.2.

4.2.2. Testing Generalization Accuracy. The main advantage a parallel system has over the conventional system used is the speed of generalization. This is due to the use of a logarithmic broadcast and gather scheme during generalization instead of the linear search necessary on a serial machine. For example, in order to do any of the Distance Metric generalization styles, the k nearest prototypes to the input vector $\boldsymbol{I}$ must be found. On a serial machine this is done with a linear search over all of the prototypes, but on a parallel architecture this can be done as follows.

First, $\boldsymbol{I}$ is broadcast to all nodes in the prototype tree. This is done by presenting the instance E to the root node and allowing each non-empty node receiving E to pass it to both its children.

Next, each non-empty node N waits for both of its children to respond with either the signal "done", or with the prototype with the smallest distance they have found in their subtrees. (All nodes with empty prototypes simply signal "done" to their parents.)

Once N hears from both its children, it repeatedly signals its own parent in one of three ways:

(1) If N's distance is better (smaller) than either of the distances being reported by its children, then N's distance and prototype are sent up to its parent, and its distance ($N.d$) is set to "done."

(2) Otherwise, if the left or right child has the smallest distance, then the distance and prototype it is reporting is passed up to the parent, and that child is told to present another distance and prototype.

(3) If N and both its children are "done", then "done" is signaled to the parent.

Once N signals its parent with a distance and a prototype, it waits to hear that its parent has used that prototype and wants another one. The process is repeated until either N and both of its children are all "done" (i.e., the whole subtree has been sent to N's parent), or another example E is broadcast down from above (i.e., the root has gotten all the prototypes it needed and has moved on to the next input vector).

In this way, each node presents as many prototypes as requested from its subtree, all sorted from closest to farthest. The environment therefore can pull the k closest prototypes from the root node and then do voting and subsequent generalization from there.

4.3. Efficiency Considerations

On the serial computer, the process of creating the prototype tree takes $O(n\log n)$ time and uses $O(n)$ storage, where n is the number of instances in the training set. Computing the conflict can be done in $O(n)$ time because all the prototypes with the same inputs are adjacent in an inorder traversal of the tree. Generating and storing the first-order features takes approximately $O(mn\log m)$ time, where m is the number of input variables in an instance. This step generates a number of first-order features which depends more on the number of inputs than on the number of instances in the training set. There will never be more than $m*n$ features, but in practice there are far fewer than that—usually between m and m^2—because there are many duplicate features. Since m is generally much smaller than n, the number of prototypes is in practice quite manageable.

On the parallel machine, creating the prototype tree takes $O(n\log n)$ time and creating the feature tree takes $O(mn\log m)$ time—almost the same as the serial method.

Testing the generalization accuracy of each style on the test set can be the most time-consuming step on the serial machine because it takes $O(snm)$ time (s styles, n instances with m variables looked at for the distance metrics) for each instance to test generalization for the various styles. Therefore, using all n available instances as a test set requires $O(sn^2m)$ time. For large training sets this can be quite restrictive, but for large training sets all n instances often do not need to be used, because an acceptable statistical sample often can be obtained from a subset of the available instances, regardless of the actual size of the training set. Subsequent generalization on a serial machine would take $O(nm)$ time for distance metrics.

On the parallel system, generalization takes $O(k+\log n)$ time, because it is possible to use a broadcast and gather scheme as discussed in Section 4.2, and operations can be pipelined such that the k closest prototypes can be retrieved in $O(k)$ time steps once the first one is ready.

4.4. Handling Continuously Valued Data

The current implementation of these prototype styles of generalization assumes multi-state input and output values. Since many databases have continuously valued input values, these continuously valued variables must be converted into multi-state variables before learning and generalization can take place. This is done with an *equal frequency method*, by doing a histogram equalization on the continuously valued input values for each variable and dividing the resulting distribution into a certain number of states. Several numbers of states were tried, and using ten states seemed to yield reasonable results. It is possible that the number of states chosen and the clustering algorithm used in this preprocessing could have a great effect on the subsequent generalization ability of the above styles, and current research is addressing this question (Ventura 94).

4.5. Unknown Inputs

Some instances in some of the databases used have unknown input values. One way to handle unknown input values is to treat them as "don't care" variables, in which case they are considered to match any other value. Problems arise with this method, however, because instances with many unknown input values are too often considered better matches than instances with asserted values. Therefore this scheme favors incomplete (and possibly unreliable) instances.

One illustrative example of this problem occurrs in the "House Votes" database, which contains one instance which is made up entirely of unknown inputs. This instance matches every variable on every other instance, which essentially wipes out all other instances' influence unless there is a direct match somewhere.

A second way to handle unknown inputs is to treat unknown inputs as mismatches in all cases. This method clearly treats instances with unknown inputs harshly but in general is much more reliable in terms of accuracy than the former option. This method, therefore, was used for the simulations which produced the results presented below.

Another possible way to handle unknown inputs is to treat them as simply another value, so that unknown inputs match other unknown inputs and nothing else. In some cases, where not knowing a value tells something specific about the problem, this may be helpful.

Unknown inputs can be normalized in the following manner. If a prototype has m input variables, u of which have unknown input values, then distances can be normalized such that the unknown inputs do not count for or against the instance, i.e., consider the distance for the unknown input to be 0, and then multiply the final summed distance by

$$\frac{m}{m-u},\ m \neq u \qquad \text{(Eq. 4.2)}$$

If m=u, then the distance can be set (arbitrarily) to m, because all the inputs are unknown.

Example 4.1. Consider the input vector I and the two prototypes, P_1 and P_2, shown in Figure 4.2, below.

Instance: 1 1 1 1	Hamming Distance	* "Unknown" Normalization		Normalized Distance
P1: 1 1 0 0	HD(P1.**I**,**I**)	$* \frac{m}{m-u}$ =	(0+0+1+1) $* \frac{4}{4-0}$	= 2 * 1 = **2**
P2: 1 ? 0 ?	HD(P2.**I**,**I**)	$* \frac{m}{m-u}$ =	(0+Ø+1+Ø) $* \frac{4}{4-2}$	= 1 * 2 = **2**

Figure 4.2. Normalizing distance metrics for prototypes containing unknown inputs.

The Hamming Distances for P_1 and P_2 are 2 and 1, respectively. In each case half of the known inputs are mismatched, so normalizing the two distances makes them the same.

Going back to the previous methods mentioned above, if unknowns were considered mismatches, then the Hamming Distance for P_2 and $\boldsymbol{I}$ would be 3, since only one variable is really matched. If, on the other hand, unknowns were considered to be matches, then the distance would be 1, since only the third variable is definitely wrong.

Chapter 5
Empirical Results and Analysis

The generalization styles presented in this thesis were implemented as described above and tested on each of the databases listed in Section 5.1. The results are encouraging, for they suggest that prototype styles of generalization can be effective on many applications. Furthermore, they support the hypothesis that no one style generalizes best on all applications, and that having a collection of generalization styles to choose from is helpful.

5.1. Machine Learning Databases

The data used for these simulations was taken from the collection of Machine Learning Databases at the University of California Irvine (Murphy 93). A short description of each of the applications used is given below.

Audiology. The Audiology database was donated by Ross Quinlan, and takes a large number of Boolean or multi-state attributes as input, and returns a diagnosis of an audiological disorder (ear disease) as output. The output has 24 possible values.

Wisconsin Breast Cancer. This database was obtained originally from the University of Wisconsin Hospitals, Madison from Dr. William H. Wolberg (Wolberg 1990). The goal is to determine whether a tumor is benign or malignant, given nine 10-state linear input variables.

Credit Rating. This database has had all attribute names and values changed for the purpose of confidentiality, but includes a mixture of continuously valued data and nominal (non-linear) multi-state variables. The purpose is to decide whether or not someone should be considered a good enough "risk" to be given a loan.

Glass. Ten continuously valued variables (converted to 10-state multi-state variables for the simulations) are used to classify what the resulting type of glass could be used for.

Hepatitis. This database uses several Boolean inputs and linear (but discrete) inputs to predict whether or not a hepatitis patient will die.

House Votes. The voting record of congressmen is used to indicate whether that person is a Democrat or Republican. This database contains many "unknown" values, including one instance that has only unknown input values.

Iris. Three classes of iris plants are predicted on the basis of sepal and petal width and length.

Lung Cancer. This database has 56 (4-state) input variables, but only 32 instances, so it is extremely difficult for a learning algorithm to learn enough to predict which variables are important.

Nettalk. Given seven consecutive letters, the correct phoneme and accent to use on the middle letter is predicted. The remaining six letters are used for context.

Pima Indians Diabetes. Uses the condition and history of the patient to predict whether the person tested positive for diabetes.

Segmentation. Several images were used, and 19 attributes were extracted from parts of them. The goal is to decide what texture (e.g., brick face, sky, grass, etc.) is located at that part of the image.

Shuttle Landing. The Shuttle Landing Control database is not really a training set at all. It is a set of carefully constructed rules, much like the original ASOCS models are designed for, which decide how to proceed with a space shuttle landing given various whether conditions. This database was included to indicate how distance metrics might work on rule-based instance sets.

Sonar. Inputs extracted from sonar blips are used to determine whether a submarine is detecting a rock or a mine.

Soybean (large). The soybean databases contain 35 nominal attributes describing the condition of soybean plants, and the goal is to determine what (if any) disease they are suffering from. The instances can be put into 19 possible classes.

Soybean (small). The "small" soybean database contains only 47 of the original 307 instances from the "large" soybean database and has only 4 possible classes for the output.

Vowel Recognition. Given ten continuously valued inputs representing the sound a person made, the problem is to decide which of 10 vowel sounds was spoken.

Zoo. Given several attributes about an animal (e.g., whether it has fur, feathers, etc.), the animal must be classified as a mammal, bird, etc.

5.2. Distance Metrics and First-order Features

An overview of the results involving the distance metrics and first-order features appears in the 3-dimensional graph in Figure 5.1. The heights in the graph indicate the percentage of accurate generalization achieved by the generalization style listed on the bottom when used on the application shown on the right. Left of all the generalization styles is an entry called "Max of all styles" which summarizes the maximum achieved by any of the styles in the graph. The applications are sorted from lowest to highest in terms of maximum generalization accuracy.

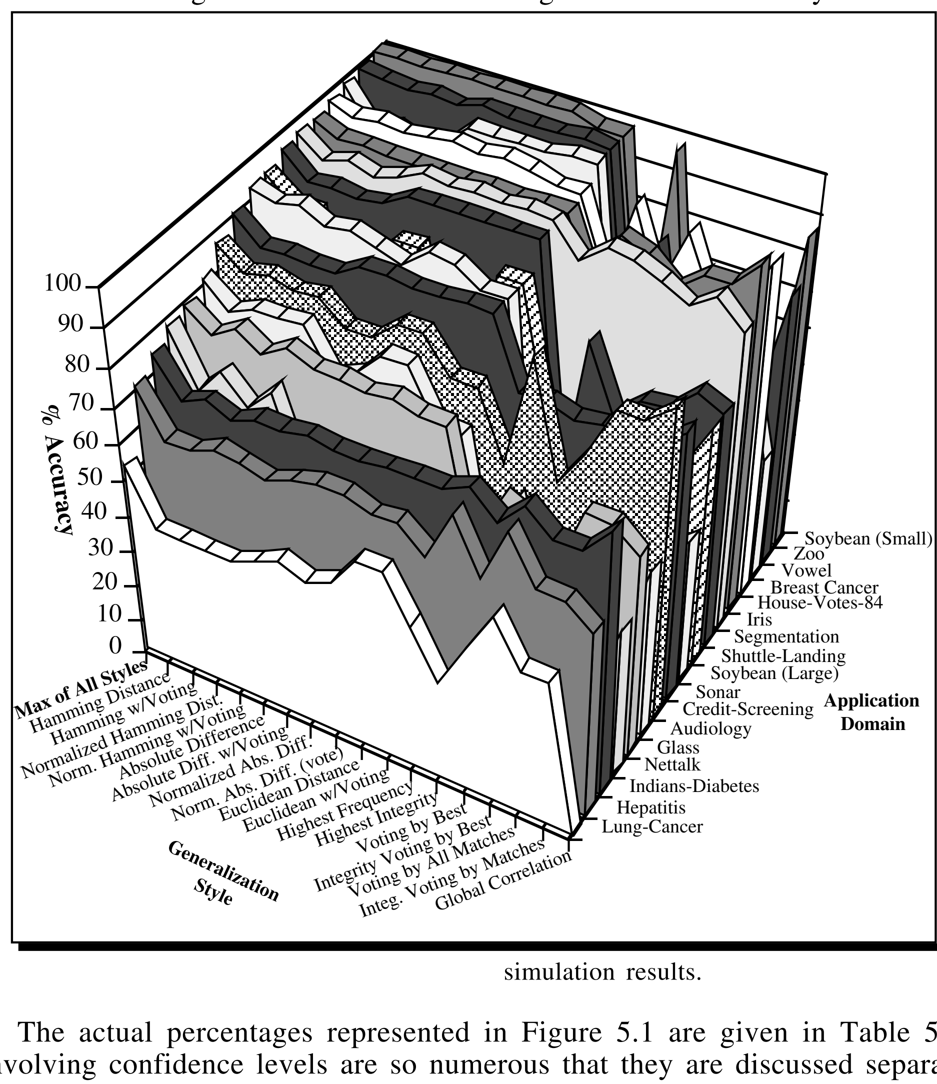

Figure 5.1. Overview of simulation results.

The actual percentages represented in Figure 5.1 are given in Table 5.2, below. The styles involving confidence levels are so numerous that they are discussed separately. The percentages shown indicate the generalization accuracy of each style when used on the application listed on the left. Those entries which appear in bold type indicate the highest percentage achieved for each application by any of the styles shown in the table. The column labeled "Max of All Styles" contains the maximum generalization accuracy of any of the styles for each application, including the confidence level styles which are not shown in the table. This means that a few applications (e.g., Glass, Nettalk) have higher values in the "Max" column than any of the values to the right, indicating that a confidence level style did better than any of those in the table. The applications are ordered from best to worst in terms of maximum generalization ability.

		Distance Metrics										**First-order Features**						
Database	**Max of all Styles**	Hamming Distance	HD w/voting	Normalized HD	NHD w/voting	Absolute Difference	AD w/voting	NAD	NAD w/voting	Euclidean Distance	ED w/voting	Highest Count	Highest Integrity	Voting by *Best*	Integ. Voting by *Best*	Voting by *Match*	Integ.Voting by *Match*	Highest Correlation
Soybean (Small)	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	98	98	36	98	36	64	36	36	87
Zoo	**99**	98	97	98	97	**99**	98	98	98	**99**	98	41	78	41	62	41	41	76
Vowel	**98**	89	91	89	91	97	97	97	97	**98**	**98**	40	40	43	34	53	65	35
Breast Cancer	**97**	95	96	95	96	**97**	96	**97**	96	94	94	74	96	77	93	77	80	92
Iris	**96**	91	93	91	93	**96**	95	**96**	95	**96**	95	87	93	92	89	85	90	83
House-Votes-84	**96**	93	93	93	93	93	93	93	93	93	93	76	92	89	83	89	89	**96**
Segmentation	**95**	89	90	89	90	**94**	**94**	**94**	**94**	**94**	**94**	28	73	56	62	63	69	66
Soybean (Large)	**93**	91	92	90	91	89	87	92	91	87	87	0	58	2	46	29	35	41
Shuttle-Landing	**93**	**87**	**87**	40	40	**87**	**87**	53	53	**87**	**87**	53	53	53	53	53	53	67
Sonar	**90**	84	84	84	84	**88**	**88**	87	87	**88**	**88**	70	70	68	69	70	70	75
Credit-Screening	**86**	81	83	81	82	76	76	82	82	73	73	56	**86**	56	66	80	80	**86**
Audiology	**81**	75	78	78	**79**	68	71	78	78	65	65	24	53	24	44	24	24	48
Glass	**78**	73	**77**	73	**77**	74	74	73	74	72	72	37	58	55	44	63	63	57
Nettalk	**77**	63	**70**	63	**70**	48	48	48	48	42	43	17	42	17	40	19	21	42
Indians-Diabetes	**74**	64	67	64	67	67	68	67	67	68	68	68	71	65	72	66	66	**74**
Hepatitis	**70**	58	58	61	60	58	61	63	63	60	61	55	69	55	**70**	58	58	52
Lung-Cancer	**53**	38	38	38	38	41	44	41	44	**53**	**53**	41	28	41	**53**	41	41	0

Table 5.2. Percentages of accurate generalization.

There are several interesting things to note from this table.

5.2.1. *The Need for Multiple Styles.* First of all, note that there no one style generalized most accurately on all applications. The table indicates this by the fact that no styles have a column which is entirely bold. This suggests that by using a collection of generalization styles and picking the one that did the best on the test set, better generalization can be achieved than can be by using only one style.

5.2.2. *Distance Metrics vs. First-order Features.* The distance metrics seemed to do better than the first-order features on most applications. This makes sense for several reasons. First of all, first-order features look at only one variable's correlation with the output. They ignore higher-order relationships that often are vital to the application. Secondly, the integrity of features often is based on an insufficient sample size. That is, the integrity of each feature is based on the feature's frequency and conflict. If these are small numbers, then the ratios are going to vary drastically with small fluctuations in the values and the integrity will be fairly unreliable.

On the other hand, the critical feature styles did as well or better than the distance metrics on a few applications, especially those that seemed hard for the distance metrics. In the Indians Diabetes database, for example, one of the variables was 74% correlated with the output value, so using the Highest Correlation style, that accuracy was achieved, while none of the distance metrics were able to do better than 68%. The other first-order feature styles did well, too, because all of the variables had at least a 65% correlation with the output. In the House Votes database, there was one variable which had a 96% correlation with the output, so the Highest Correlation style was able to do better than the distance metrics on that application as well.

In contrast, the Vowel database did not have any input variables that had a correlation of more than 37% with the output, so the Highest Correlation style—as well as the other first-order feature styles—did quite poorly, while the distance metrics were able to achieve 98% accuracy. This indicates the importance of higher-order relationships in some applications.

5.2.3. *The Effect of Voting.* As expected, the distance metric styles of generalization that used voting did as well or better than their corresponding non-voting style in almost every case. In the Nettalk database, for example, Hamming Distance alone generalized with 63% accuracy, while the addition of voting raised the accuracy to 70%. In the Zoo database, on the other hand, the non-voting style of Hamming distance did slightly better than the voting style. When voting is not used, the first of the prototypes which are equally close to the input is chosen arbitrarily to decide the output. In this case, then, the arbitrary ordering of the prototypes just happened to favor the correct output slightly more than it favored incorrect outputs. The difference is not statistically significant, however, so it is fairly safe to assume that voting will not in general hurt generalization accuracy and improves accuracy for some applications.

5.2.4. *Similarities Among Distance Metrics.* There are many applications for which the various distance metrics have a very similar accuracy. This is not surprising, for in some cases the various styles are equivalent. For example, if all the input variables have the same number of states, then normalizing the Hamming distance or absolute difference will not have any effect. This is the case with the Breast Cancer database, for example.

In addition, if the input variables are all Boolean, then Hamming Distance, Absolute Difference, and Euclidean Distance are all equivalent, because the distance for each variable will always be 0 or 1, and squaring this distance still returns 0 or 1. In the House Votes 84 database, for example, all the input variables are Boolean, and all of the distance metrics have the same accuracy. It would be possible for the voting and non-voting columns to have two separate values, but in this case they happen to be equal.

5.2.5. *Differences Among Distance Metrics.* There are several factors which contribute to differences among the various distance metrics. Voting has the effect of breaking ties between equally close prototypes in a non-arbitrary way and thus usually has a positive influence on generalization accuracy. In addition, the nature of the application itself has a significant effect on which style or styles of generalization will work best. Using Absolute Difference or Euclidean Distance on input variables with nominal (non-linear) input values makes little sense. In the Soybean databases, for example, all the input variables are nominal, and Euclidean Distance performs more poorly than Hamming Distance because it tries to place importance on the relative distance between nominal values.

On the other hand, the Segmentation database has linear values for all its input variables, and Hamming Distance does not perform as well as Absolute Difference or Euclidean Distance, because the actual distance *is* important in this application.

5.3. Confidence Level Extensions

There are too many generalization styles involving confidence level extensions for them to be listed in Table 5.2. Section 3.8 presented six ways of computing a confidence level (CL) for influencing voting power (i.e., PI, BF, AF, OF, WOF and WAF}. Each of these could be used alone or run through one of the two squashing functions presented in Section 3.11 (DC and CL^2). Furthermore, five different distance metrics are used on each of those combinations (i.e., HD, NHD, AD, NAD and ED). This results in 6x3x5=90 confidence level styles of generalization, in addition to the 2x5=10 distance metric styles listed in Table 5.2 (i.e., frequency voting or no voting, for each of the five distance metrics). Furthermore, the distance measures themselves can be influenced by the several distance confidence levels (DCL) mentioned in Section 3.9 (i.e., COR, PF and BF). This results in quadrupling the number of distance metric styles to a total of 400.

Since the number of styles involving confidence levels is so large, an overview of the results and general trends is given without listing the actual percentages for all the CL styles on all the applications. However, the results of a representative sampling of the confidence level styles are given in Table 5.3 below. At the top of each column, each generalization style is defined by

- The distance metric used (HD, NHD, AD, NAD, or ED)
- The CL used to weight voting (PI, BF, AF, OF, WOF, or WAF)
- The DCL used to weight distance (COR, PF, or BF)

and some of the confidence levels are modified with the double cosine (DC) squashing function. The word "none" indicates a weight of 1.0 is applied instead of a confidence level.

	Max of all Styles										
		none	COR	DC(COR)	DC(COR)	none	BF	PF	COR	COR	**Distance Weight**
		OF	none	BF	BF	WAF	none	none	DC(OF)	none	**Voting Confidence Level**
Database		HD	HD	HD	NHD	AD	AD	AD	NAD	ED	**Basic Distance Metric**
Soybean (Small)	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	
Zoo	**99**	97	97	97	97	98	68	89	98	98	
Vowel	**98**	90	91	89	89	97	94	97	97	**98**	
Breast Cancer	**97**	96	95	95	95	**97**	75	74	96	94	
Iris	**96**	93	91	91	91	94	94	90	95	95	
House-Votes-84	**96**	92	94	94	94	92	46	61	94	94	
Segmentation	**95**	90	90	89	89	94	52	62	**95**	94	
Soybean (Large)	**93**	91	**93**	90	90	89	49	88	91	87	
Shuttle-Landing	**93**	**93**	**93**	**93**	47	87	60	47	**93**	**93**	
Sonar	**90**	83	84	84	84	87	88	**90**	87	88	
Credit-Screening	**86**	82	81	81	80	77	66	66	82	73	
Audiology	**81**	75	77	77	**81**	68	16	45	78	66	
Glass	**78**	**78**	74	73	73	76	26	40	76	74	
Nettalk	**78**	66	77	**78**	**78**	48	17	25	59	47	
Indians-Diabetes	**74**	68	65	65	65	67	46	43	66	68	
Hepatitis	**70**	60	59	61	61	62	48	63	61	61	
Lung-Cancer	**59**	38	34	37	37	44	**59**	53	47	41	

Table 5.3. Accuracy of selected confidence level styles.

5.3.1. *Confidence Levels Weighting Votes.* The overall performance of the confidence level styles of generalization involving voting was quite similar to that of the regular distance metrics, as is to be expected; voting is used only when two or more prototypes are equally close to the input. When voting is not needed, as it often is not, the confidence levels used to influence voting power have no effect.

Confidence levels helped voting slightly in several applications. In the Glass database, for example, weighting the votes by the weighted overall integrity of the prototype's features increased accuracy by 1.5% on Hamming Distance, and resulted in the highest accuracy of any of the styles used. This is why a bold "78" appears in the "Max" column while the highest accuracy shown in the table for the Glass database is 77%.

There were times when the squashing functions helped generalization slightly (usually by less than 1%) and times when they hindered it by about the same amount, but the effect was often quite small.

5.3.2. *Confidence Levels Weighting Distances.* Since distances are used during every classification, the confidence levels that weighted distances had a somewhat greater effect on generalization accuracy. For example, in the Sonar database, multiplying the distance for each variable by the integrity of the prototype's corresponding feature for that variable improved accuracy by almost 3% when using Absolute Difference as the distance metric. The reason this helped on this application is due likely to the fact that most of the variables had a decent correlation with the output (at least 50% for all of them with some up to 75% correlation).

In the Nettalk database, multiplying the Hamming distance for each variable by the overall correlation improved accuracy from 70% to 77%—a significant improvement. The reason for this improvement is clear if one looks closely at the format of the instances. Each instance contains a seven-letter “window” into a word, and the middle (fourth) letter is the one for which the proper pronunciation is sought. Clearly, the middle letter is the most important for the classification, and the other letters are progressively less important the further they are from the center.

The correlation of the seven input variables reflects this relationship, as shown in Figure 5.3, below. As expected, the middle input variable has a much higher correlation with the output than the “context” letters.

Preceeding Letters			Letter to be	Trailing Letters		
-3	-2	-1	Classified	+1	+2	+3
11%	14%	16%	42%	18%	17%	15%

Figure 5.4. Correlation of Nettalk’s input variables with the output.

Even though the middle input variable has a much higher correlation than the other variables, a 42% correlation is still too low to be of much use to the first-order features. In fact, none of them generalized better than 42% on Nettalk, as can be seen in Table 5.2.

The confidence levels that weighted distances did not always perform well, however. Consider the Nettalk database again. Weighting the distances by the integrity of the prototype’s corresponding features resulted in accuracies of less than 25% because the individual features were so unreliable.

In fact, in almost every case weighting the distance by overall correlation did better than using the prototype’s individual features to weight the distance. This may be due to the fact that the overall correlation is based on a larger sample size and is thus more reliable in determining which variables are most important. When the first-order features were based on too small of a sample size or were not correlated well with the output, generalization was often quite poor when distances were weighted by prototypes’ features. Lowering accuracy by 20-30% was not uncommon in such cases.

Database	**Maximum**	Distance Metrics	Confidence Levels	First-order Features	CTree (C4.5)	CRule (C4.5)	ID3	C4	IB3	IB4
Soybean (Small)	**100**	**100**	**100**	**100**	98	98	98	61	**100**	**100**
Shuttle-Landing	**100**	87	93	67	**100**	**100**	**100**	**100**	99	99
Sonar	**100**	88	90	75	75	74	77	72	66	71
Zoo	**99**	**99**	98	78	93	92	98	93	93	93
Vowel	**98**	**98**	**98**	65	80	77	83	80	94	94
Breast Cancer	**97**	**97**	**97**	96	95	89	95	94	95	95
Iris	**97**	96	96	93	95	95	94	95	95	97
House-Votes-84	**97**	93	94	96	96	96	95	97	92	95
Segmentation	**97**	94	95	73	96	89	**97**	96	94	94
Indians-Diabetes	**93**	68	68	74	73	68	70	74	69	70
Soybean (Large)	**93**	92	**93**	58	90	87	90	92	89	90
Credit-Screening	**86**	83	83	**86**	82	73	79	83	82	83
Hepatitis	**84**	61	61	70	80	**84**	78	78	75	78
Audiology	**81**	79	**81**	53	77	78	78	77	67	69
Glass	**78**	77	**78**	63	68	67	69	67	64	68
Lung-Cancer	**59**	53	**59**	53	43	41	47	36	36	39

Table 5.5. Comparison between prototype styles of generalization and other models.

5.4. Comparison with Other Machine Learning Models

Table 5.5 summarizes a comparison between the generalization styles presented in this thesis (referred to as the *prototype styles*), and several well-known machine learning models. The maximum accuracy achieved by the basic distance metrics, the confidence level extensions, and the first-order features are given in the center three columns. Some results obtained using 10-fold cross-validation for C4, C4.5 ("CTree" and "CRule"), ID3, IB3 and IB4 (Zarndt 94) are listed on the right.

The results for the prototype styles were obtained using N-fold cross-validation, which may help accuracy slightly. As with previous tables, bold entries indicate the highest accuracy achieved by any of these styles for each application.

Again no single style has the highest accuracy on every database. Also note that the prototype styles of generalization were able to generalize at least as well as these other well-known machine learning systems on a variety of applications. The prototype styles did significantly better on a few applications (e.g., Sonar and Glass), but they were were significantly less accurate on others (e.g., Shuttle-landing and Hepatitis).

5.5. The Importance of Using Multiple Styles

The results clearly indicate that it having more than one style of generalization available for solving a problem is advantageous. Among the prototype styles of generalization presented in this thesis, no single style had (or tied for) the highest generalization accuracy for more than 4 out of the 17 applications used in the simulations. Several styles came within 5% of the best generalization on 12 out of the 17 applications, but no styles were even within 10% of the best accuracy on all 17 of them. In other words, no matter which single style one selects, there would be some applications on which that style would not be able to come within 10% of the accuracy of one of the other styles. This makes it fairly clear that no one of these styles of generalization would be sufficient on its own.

5.6. Summary of Findings

The results presented above support the intuition that Hamming Distance works well on input variables with nominal (non-linear) values but that Absolute Difference or Euclidean Distance work better on linear input variables. These distance metrics are all equivalent when the inputs are all Boolean.

Furthermore, first-order features appear to be of some use when there are one or more variables which correlate well with the output. In particular, the Highest Correlation method works best when there is one variable which is correlated very highly with the output, and the other first-order features seem to do well when many of the variables have a fairly good correlation with the output.

Voting tends to improve, or at least does not impair, the generalization of distance metrics in almost all cases. Normalizing seems to help when there are wide differences in the number of states of linear input variables but has no effect when all inputs have the same number of states.

The confidence levels have a fairly small effect when applied to voting power, but seem capable of improving generalization by a few percent at times, especially when the first-order features are based on a large enough sample size to be trustworthy. Using confidence levels on voting power rarely lowers generalization accuracy by more than one or two percent.

Weighting distance measures with the overall variable correlation is capable of improving generalization significantly. This is especially true if one variable stands out as being much more important than the others but can impair accuracy when this is not the case. Furthermore, weighting distance measures with the integrity values of a prototype's corresponding first-order features only seems to help when the features are trustworthy and have a decent correlation with the output. Otherwise, this method often performs quite poorly, and it almost never does better than using the Highest Correlation for weighting distances.

Chapter 6
Conclusions and Future Research Areas

The results of this research suggest that using prototype styles of generalization can provide efficient and accurate generalization for some applications without the need for iteration, numerous “system parameters”, exponential searches or storage, or potential for getting stuck in local minima. In addition, the results support the hypothesis that using multiple styles of generalization can be more effective than using any one style by itself.

Furthermore, future research may be able to provide automated ways of not only picking one style of generalization which is most accurate for a particular application but also making use of a collection or “committee” of several styles of generalization at the same time. Using such a method, the collection of guesses made by the various styles of generalization could themselves be used to predict the correct output more often than any of the individual styles would be able to do (Wolpert 93).

In addition, separate styles of generalization could be used on each individual input variable. For example, Hamming Distance could be used on nominal inputs, while normalized Euclidean Distance could be used on continuously valued variables.

The ultimate goal of this research is to build up a good collection of generalization styles and then construct systems which automatically determine the generalization style or styles that work best with any given application, thus providing fast, accurate solutions to applications for which automated solutions are too costly or unknown.

Bibliography

Aha, David W., Dennis Kibler, Marc K. Albert, "Instance-Based Learning Algorithms," *Machine Learning*, vol. 6, pp. 37-66, 1991.

Clark, Peter, and Tim Niblett, "The CN2 Induction Algorithm," *Machine Learning*, vol. 3, pp. 261-283, 1989.

Cover, T. M., and P. E. Hart, "Nearest Neighbor Pattern Classification," *Institute of Electrical and Electronics Engineers Transactions on Information Theory*, vol. 13, pp. 21-27, 1967.

Dasarathy, Belur V., and Belur V. Sheela, "A Composite Classifier System Design: Concepts and Methodology," *Proceedings of the IEEE*, vol. 67, no. 5, pp. 708-713, May 1979a.

Dasarathy, Belur V., and Belur V. Sheela, "Design of Composite Classifier Systems in Imperfectly Supervised Environments," *Proceedings of the Conference on Pattern Recognition and Image Processing*, pp. 71-79, August 1979b.

Hart, P. E., "The Condensed Nearest Neighbor Rule," *Institute of Electrical and Electronics Engineers Transactions on Information Theory*, vol. 14, pp. 515-516, 1968.

Koton, P., "Reasoning About Evidence in Causal Explanations," *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 256-261. St. Paul, MN: Morgan Kaufmann, 1988.

Lippmann, Richard P., "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, vol. 3, no. 4, pp. 4-22, April 1987.

Martinez, Tony R., "ASOCS: Towards Bridging Neural Network and Artificial Intelligence Learning," Presented at *2nd Government Neural Network Workshop*, 1991.

Martinez, Tony R., Douglas M. Campbell, Brent W. Hughes, "Priority ASOCS," to appear in *Journal of Artificial Neural Networks*, 1994.

Martinez, Tony R., "Adaptive Self-Organizing Concurrent Systems," *Progress in Neural Networks*, vol. 1, ch. 5, pp. 105-126, O. Omidvar (Ed), Ablex Publishing, 1990a.

Martinez, Tony R., "Consistency and Generalization in Incrementally Trained Connectionist Networks," *Proceedings of the International Symposium on Circuits and Systems*, pp. 706-709, 1990b.

Martinez, Tony, "Neural Network Applicability: Classifying the Problem Space," *Proceedings of the IASTED International Symposium on Expert Systems and Neural Networks*, pp. 12-15, 1989.

Martinez, Tony R., "Adaptive Self-Organizing Logic Networks," Ph.D. Dissertation, University of California Los Angeles Technical Report - CSD 860093, (274 pp.), June, 1986.

Michalsky, R. S., I. Mozetic, J. Hong, and N. Lavrac, "The Multi-purpose Incremental Learning System AQ15 and its Testing Application to Three Medical Domains," *Proceedings of the Fifth National Conference on Artificial Intelligence*, pp. 1041-1045, Philadelphia: Morgan Kaufmann, 1986.

Michalsky, R. S., and J. Larson, "Incremental Generation of VL_1 Hypotheses: The Underlying Methodology and the Description of the Program AQ11", *Technical Report ISG 83-5*, Urbana: University of Illinois, Computer Science Department, 1983.

Michalsky, R. S., "On the quasi-minimal solution of the general covering problem," *Proceedings of the Fifth International Symposium on Information Processing*, pp. 12-128. Bled, Yugoslavia, 1969.

Murphy, P. M., and D. W. Aha, *UCI Repository of Machine Learning Databases*. Irvine, CA: University of California Irvine, Department of Information and Computer Science, 1993.

Porter, Bruce W., Ray Bareiss, and Robert C. Holte, "Concept Learning and Heuristic Classification in Weak-Theory Domains," *Artificial Intelligence*, vol. 45, nos. 1 and 2, 1990.

Quinlan, J. R., *C4.5: Programs for Machine Learning*, San Mateo, CA: Morgan Kaufman, 1993.

Quinlan, J. R., "Simplifying Decision Trees," *International Journal of Man-Machine Studies*, vol. 27, pp. 221-234, 1987.

Quinlan, J. R., “Induction of Decision Trees”, *Machine Learning*, vol. 1, pp. 81-106, 1986.

Rissland, E. L., J. Kolodner, and D. Waltz, “Case-based reasoning” *from* DARPA: Machine Learning Program Plan, *Proceedings of the Case-Based Reasoning Workshop*, pp. 1-13. Pensecola Beach, FL: Morgan Kaufmann, 1989.

Rudolph, George L., and Tony R. Martinez, “An Efficient Static Topology For Modeling ASOCS,” *International Conference on Artificial Neural Networks*, Helsinki, Finland. In *Artificial Neural Networks*, Kohonen, et. al. (Eds), Elsevier Science Publishers, North Holland, pp. 729-734, 1991.

Rumelhart, D. E., and J. L. McClelland, *Parallel Distributed Processing*, MIT Press, Ch. 8, pp. 318-362, 1986.

Salzberg, S., “A Nearest Hyperrectangle Learning Method,” *Machine Learning*, vol. 6, pp. 277-309, 1991.

Stanfill, C., and and D. Waltz, “Toward memory-based reasoning,” *Communications of the ACM*, vol. 29, pp. 1213-1228, 1986.

Ventura, Dan, and Tony R. Martinez, “BRACE: A Paradigm For the Discretization of Continuously Valued Data,” to appear in *Proceedings of the Seventh Annual Florida AI Research Symposium–(FLAIRS)*, 1994.

Wettschereck, Dietrich, and Thomas G. Dietterich, “An Experimental Comparison of Nearest-Neighbor and Nearest-Hyperrectangle Algorithms,” *to appear in Machine Learning*, 1994a.

Wettschereck, Dietrich, “A Hybrid Nearest-Neighbor and Nearest-Hyperrectangle Algorithm”, *To appear in the Proceedings of the 7th European Conference on Machine Learning*, 1994b.

Widrow, Bernard, and Rodney Winter, “Neural Nets for Adaptive Filtering and Adaptive Pattern Recognition,” *IEEE Computer Magazine*, pp. 25-39, March 1988.

Wilson, D. Randall, and Tony R. Martinez, “The Importance of Using Multiple Styles of Generalization,” *Proceedings of the First New Zealand International Conference on Artificial Neural Networks and Expert Systems (ANNES)* , pp. 54-57, November 1993.

Wilson, D. Randall, and Tony R. Martinez, “The Potential of Prototype Styles of Generalization,” *The Sixth Australian Joint Conference on Artificial Intelligence (AI ’93)*, pp. 356-361, November 1993.

Widrow, Bernard, and Michael A. Lehr, “30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation,” *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1415-1441, September 1990.

Wolberg, William H., and O.L. Mangasarian, “Multisurface Method of Pattern Separation for Medical Diagnosis Applied to Breast Cytology", *Proceedings of the National Academy of Sciences*, vol. 87, pp. 9193-9196, December 1990.

Wolpert, David H., “Combining Generalizaers Using Partitions of the Learning Set,” To appear in *1992 Lectures in Complex Systems*, L. Nadel and D. Stein (Eds.), Addison-Wesley, 1993.

Prototype Styles of Generalization

D. Randall Wilson

Department of Computer Science

M. S. Degree, August 1994

ABSTRACT

A learning system can generalize from training set data in many ways. No single style of generalization is likely to solve all problems better than any other style, and different styles work better on some applications than others. Several generalization styles are proposed, including distance metrics, first-order features, voting schemes, and confidence levels. These generalization styles are efficient in terms of time and space and lend themselves to massively parallel architectures.

Empirical results of using these generalization styles on several real-world applications are presented. These results indicate that the prototype styles of generalization presented can provide accurate generalization for many applications, and that having several styles of generalization available often permits one to be selected that works well for a particular application.

COMMITTEE APPROVAL: ________________________________
Tony R. Martinez, Committee Chairman

____
Gordon E. Stokes, Committee Member

____
David W. Embley, Graduate Coordinator