Automatic Feature Extraction in Machine Learning

A Thesis

Presented to the

Department of Computer Science

Brigham Young University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Rick Bertelsen

August 1994

ABSTRACT

This thesis presents a machine learning model capable of extracting discrete classes out of continuous valued input features. This is done using a neurally inspired novel *competitive classifier* (*CC*) which feeds the discrete classifications forward to a supervised machine learning model. The supervised learning model uses the discrete classifications and perhaps other information available to solve a problem. The supervised learner then generates feedback to guide the CC into potentially more useful classifications of the continuous valued input features. Two supervised learning models are combined with the CC creating ASOCS-AFE and ID3-AFE. Both models are simulated and the results are analyzed. Based on these results, several areas of future research are proposed.

Chapter 1

Introduction

1.1 Introduction

Inductive learning is a branch of machine learning. An inductive learning system uses examples to attempt to learn a mapping or partial function of input features to output features. A feature is used to convey the values of the examples. If the values of the output features are used in learning the function, then the inductive learning system is a *supervised learner*. Some inductive learning systems are restricted to input features with known values or *states*, and do not allow continuous valued features as input [8][14][4]. This thesis seeks to overcome these restrictions by proposing a method to create discrete input features from continuous input features.

Many problems solved by humans require the analysis of continuous or real valued inputs. In these problem, the actual continuous value appears to be used only to produce an intermediate *coding*. A coding is the process of transforming a continuous value into one value of a much smaller set of values. The resultant coding may then be used as an input to solve the larger problem. For example, consider the task of planning a day based on a weather prediction (i.e. what clothes to wear, what activities to plan, etc.). Possible features for this problem include wind, temperature, and outlook. A feature is simply a category of inputs to a problem. A person appears to code temperatures into a small set of possible codings perhaps including “hot”, “cold”, and “warm”. This coding of temperature may then be used as an input into the problem, as opposed to the actual continuous value of temperature. A person might code the continuous valued temperature of 99.5˚F into the coding “hot” and considering this coding with the other inputs into the problem, select a cool shirt to wear. Codings such as these are based on experience. Figure 1.1 shows some possible temperature values with a potential break down of codings (shown simply as vertical lines forming coding boundaries).

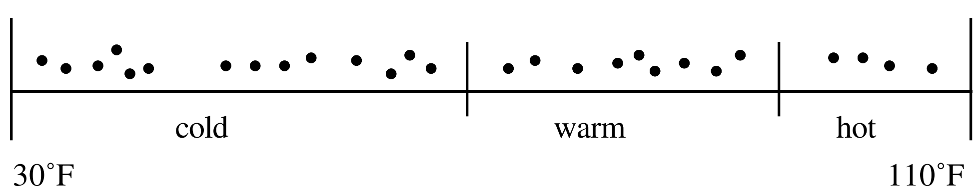


Figure 1.1 Potential temperature codings

Potential codings could be discovered spontaneously using any number of clustering algorithms [18][7]. However, the resulting codings are not guaranteed to assist in solving the problem. Clustering algorithms typically cluster according to statistical properties of the values of a feature without regard to the larger problem of inductive learning. Humans use experience with the *actual decision (the output)* of a larger problem to form useful codings. In this sense, experience *guides* a person in the formation of useful codings. Figure 1.2 shows potential improved codings derived from the initial codings given in figure 1.1. The improved codings are the result of learning, which may adjust some coding boundaries, and may even create additional codings based on the problem at hand.

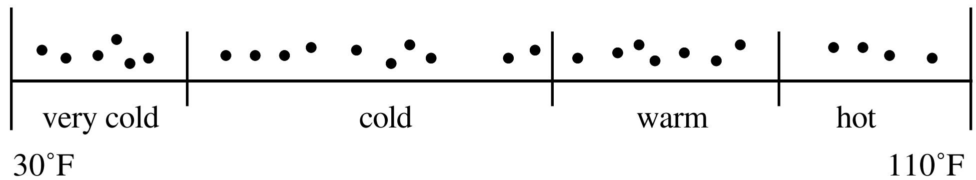


Figure 1.2 Improved codings for temperature

Coding continuous features also allows a small number of rules to apply to a much larger set of actual occurrences. This is a form of *generalization*. Generalization is the ability to extend rules to occurrences in the environment which have not been explicitly trained. By coding continuous values of temperature, a rule need not be learned for each of the possible values of temperature. Instead, one rule for the coding “hot” may replace a number of other rules.

1.2 Thesis Statement

This thesis shows how a novel competitive classifier may be combined with a supervised learning system to dynamically discover critical multi-state variables. This in turn introduces a form of generalization in the real number realm, and automatic feature extraction (AFE). The methodology is illustrated using two different supervised learning systems (referred to as SL), ASOCS [8][9][10][11] and ID3[14]. A supervised learner is a learning model which uses a known output in its learning.

ASOCS (*Adaptive Self-Organizing Concurrent Systems*) is overviewed in chapter 2 of this work. ID3 is a supervised learning algorithm which inductively builds decision trees, and is overviewed in chapter 3.

1.3 Introduction to Coding with a Competitive Classifier

Figure 1.3 shows the combination the CC with a SL. The learning model created by this combination is referred to as *SL-AFE*. In chapter 5, the SL is replaced by the specific supervised learning model creating ASOCS-AFE and ID3-AFE. The CC accepts continuous valued features as input and produces discrete codings for those features as output. The SL accepts discrete codings from the CC, and also other discrete valued features as input. Using algorithms discussed in chapters 2 and 3 the SL produces an output. The SL operates in either a *training phase*, or an *execution phase*. In the training phase, the SL compares the output it produces to a known output to generates *feedback* for the CC and perhaps makes modifications internal to the SL. Feedback is a set of commands which the CC can perform seeks to improve the codings the CC produces and are communicated to the CC through a *feedback* mechanism. The modifications of the CC and the SL attempt to make an incorrect response less likely and a correct response more likely. In the execution phase, the system acts as a hardwired circuit, and only an output is produced, there is no feedback, or any modifications made to the CC or the SL.

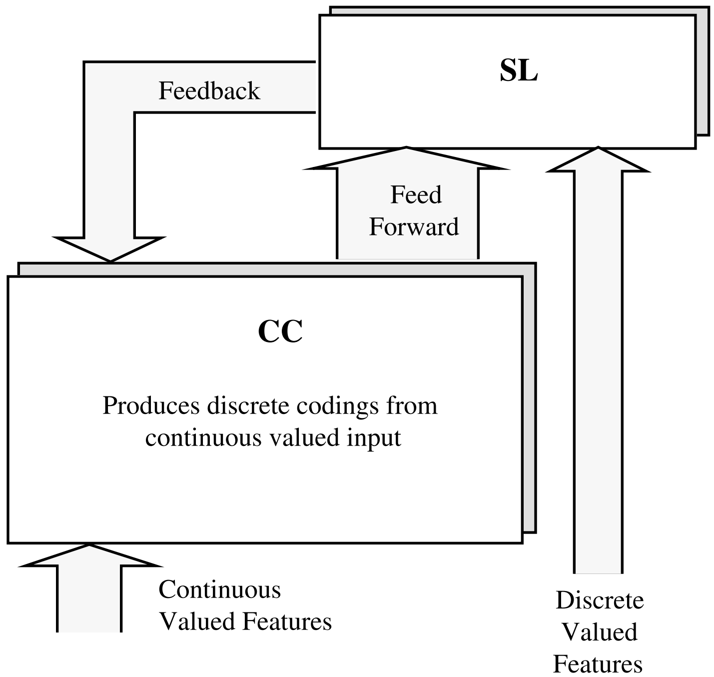


Figure 1.3 A CC combined with a SL

Figure 1.4 shows four features based on the weather example from above. Continuous valued features are shown by the word "continuous" and include the features temperature and humidity. Discrete valued features are shown with a list of the possible discrete states next to the feature name, and include the features windy and outlook. Possible states for the windy feature are true and false. States for the outlook feature are cloudy, sunny and rain.

Feature	Possible States
Temperature	Continuous
Windy	{True, False}
Outlook	{Cloudy, Sunny, Rain}
Humidity	Continuous

Figure 1.4 Input features and their possible states

The purpose of the CC is to accept continuous valued features, such as temperature, and produce discrete codings. For example, the CC may receive as input 98.5˚F, and produce the coding of "hot" as output. The codings produced are used as input to the SL. The SL uses its input to solve a given problem. Feedback is generated by the SL for the CC based on the SL's ability to solve the problem with the given inputs. Figure 1.5 shows a sample set of rules for the weather example developed above. The first rule states that *if* the temperature is 98˚F, windy is false, outlook is sunny, and humidity is 10% *then* tomorrow is a good day for golf. This rule is written in feature:value form as

Temp:98 Wind:False Outlook:Sunny Humidity:10 -> Y.

A collection of such rules is termed a *data set*. Often a data set is divided into two disjoint sets. The first is used as rules for training the SL-AFE, and is called a *training set*. The second set is used to determine if the SL-AFE learned the training set and is called a *test set*. Typically data sets such as this come from real world occurrences of a given problem.

Temp.	Wind	Outlook	Humidity	Result
98	False	Sunny	10	Y
92	True	Cloudy	35	Y
82	False	Rain	99	N
99	False	Cloudy	65	N
67	False	Sunny	50	Y
75	True	Cloudy	35	N
62	True	Rain	88	N
85	True	Sunny	15	Y
72	False	Rain	55	Y

Figure 1.5 A sample set of rules

From the first rule above in figure 1.5, the features temperature and humidity are continuous. Thus their values (98, and 10) are input to the CC in figure 1.3. The CC then uses these continuous values to form discrete codings for the features temperature and humidity. The CC might code the temperature of 98˚F into the coding “hot”, and the humidity of 10% into coding “low”. The SL uses the codings of “hot” and “low” as input, along with the discrete features of windy:false, and outlook:sunny. Based upon these four discrete features, the SL produces an output. In the training phase, the SL compares its produced output to the known output from the rule, and generates feedback based on the results of this comparison.

1.4 Contribution

The primary benefits of this thesis is to present a methodology which could extend any supervised inductive learning model which currently accepts only discrete valued input features. The methodology is demonstrated by extending the ASOCS and ID3 supervised learning models. A novel competitive classifier is developed, which can be applied as a pre-processor for any machine learning model.

1.5 Outline of Thesis

Chapter 2 of this thesis presents ASOCS and some recent extensions. Chapter 3 introduces ID3. Chapter 4 presents the competitive classifier (CC), discusses the feedback it accepts and how the CC responds to this feedback. Chapter 5 presents the combination of the CC with two different SLs, an extended ASOCS, and ID3. The generation of feedback is also discussed in chapter 5, and empirical results based on simulations is given. Chapter 6 gives conclusions and future research efforts.

Chapter 2

An Introduction to ASOCS

In the previous chapter, a methodology for solving problems is presented. Namely, using past knowledge to form codings into which the inputs of the problem are placed. Only the input features which have continuous values are coded. The discrete features are trivially grouped into proper values. This chapter examines a system developed to deal efficiently with Boolean inputs, which extended for multi-state or discrete features, becomes a basis for a problem solving system.

2.1 ASOCS

Adaptive Self Organizing Concurrent Systems (ASOCS) are developed at the University California at Los Angeles [8][9][10]. *Adaptive* implies that the structure of ASOCS changes over time. Self Organizing states that the adaptive changes to ASOCS are discovered and carried out by ASOCS, and not some external agent. *Concurrent* implies that ASOCS operates in parallel, therefore potentially faster than conventional approaches to problem solving.

2.1.1 The Instance

In order for ASOCS to adapt, some form of knowledge from the outside world must be conveyed to the system. The *instance* performs this task. An ASOCS instance is a production which contains Boolean input variables (features), and an implication for Boolean output variables when the states of the input variables are matched by the environment. An instance serves the same purpose in ASOCS as the rule discussed in chapter 1. Examples of instances are shown below.

(1) A B C -> Z

(2) A' B -> Z'

(3) A D' -> C' Z

The general form of the instance is :

Input Variables -> Output Variables.

The first example states that when the environment has set the variables A, B, and C to true, ASOCS should also set the output variable Z, to true. The second example uses the common ASOCS notation of A' to denote that variable A is set to false. Thus, this instance states that when A is false, and B is true in the environment, Z should then be set to false. Finally the third example demonstrates that ASOCS allows multiple output variables. This work, however, considers ASOCS with only a single output variable. In addition, not all possible input variables are required on the left side of expressions Thus ASOCS allows *don't care variables* in an instance and the current value of these variables is not pertinent for that instance. Finally, a variable may be used as input, output, or both.

A collection of instances is termed an *instance set* (IS). Through the IS, ASOCS defines a partial function. Inputs from the environment determine the setting of the output variable.

2.1.2 Instance Terms and Definitions

This section gives concise definitions which support subsequent sections and chapters. Each definition states a relationship between two or more instances.

Two instances are said to be *concordant* with respect to each other, if they specify the same truth setting for their output variable.

(1) A B -> Z

(2) A C' -> Z

The two instances directly above are concordant instances. Instances which are not concordant, are *discordant*. For example, if the second instance above were modified to be A C' -> Z', the above instances would be discordant. The truth setting of any given instance is termed the *polarity* of that instance. If the truth setting is true, then the instance is a *positive instance*. Likewise, if the setting is false, the instance is a *negative instance*.

Two instances are given.

(1) A -> Z

(2) A B -> Z

An instance is a *subset* of some other instance if it contains no input variable which is not an input variable in the other instance. In this example, instance (1) is a subset of instance (2). By this definition, an *equal* instance is also subset. Similarly superset is defined as in set theory. In the example above instance (2) is a superset of instance (1). Two instances which are not equal, subset, or superset are termed *overlap.*

2.2 The Instance Set

The IS represents the knowledge base of the system. Instances are added to the IS incrementally. New instances are considered to be more reliable and have precedence over older existing instances. Because of this precedence, changing the order of presentation of instances changes the final IS. Through the introduction of new instances to IS, the ASOCS adapts over time. As new knowledge is introduced, the ASOCS insures that its adaptation does not create conflicting or superfluous rules. Thus two operations are performed as each new instance is added to IS, *consistency* and *minimization*.

2.2.1 Consistency

Two instances *contradict* each other if and only if they can be simultaneously matched and they are discordant instances. A *consistent* IS has no instances which contradict. An IS which contains contradicting instances is termed *inconsistent*. An inconsistent IS is illustrated in the example below.

A B -> Z (1)

A C -> Z' (2)

Instances (1) and (2) are discordant. If variables A, B, and C are all set to true, both instances are matched, and the output variable Z could be set to either true or false. Because instances (1) and (2) contradict, any IS containing them would be inconsistent.

There are two methods used to make IS consistent: *Discriminant Variable Addition (DVA)* and *Priority*. DVA resolves the inconsistency by adding one or more variables to one of the instances which cannot simultaneously match another inconsistent instance. Priority resolves the inconsistency be setting the output variable to the output of the matching instance with the highest priority. Both methods are discussed in greater detail below.

DVA adds one or more variables to the old instance (OI) which discriminates it from the new instance (NI). A discriminant variable guarantees that the previously inconsistent instances cannot be simultaneously matched. OI then is replaced in IS by one or more variations of OI created by adding a discriminant variable to OI. Discriminant variables are found by considering the variables in NI which are not in OI. When negated, these variables can be added to OI creating a consistent IS. In the example above, assume that instance (2) is NI, and instance (1) is OI. The variable C is in NI, but not OI, and serves as the only possible discriminant variable. First negate C, and then add C' to (OI). The new IS is shown below.

A B C' -> Z (OI)

A C -> Z' (NI)

If NI contains more than one variable not in OI, each variable is used as a discriminant variable a seperate instance created for the discriminant variable. If NI contains *n* variables not referenced in OI, then *n* variations of OI will be added to IS. Each variation will contain exactly one discriminant variable. This is illustrated in the example below.

A -> Z (OI)

A B C -> Z' (NI)

Variables B and C are both discriminant variables. OI is replaced with two instances as shown below.

A B' -> Z (OI1)

A C' -> Z (OI2)

A B C -> Z' (NI)

For large instances, DVA can become expensive in terms of the number of instances created. *Priority ASOCS (PASOCS)* [12] attempts to eliminate DVA by the use of priorities. In training, when the NI is found to contradict OI, NI is simply added to IS with a greater priority than the contradicting instance. In execution, contradictions are resolved by priorities, allowing the matching instance with the highest priority to assert its output. The difference in the resulting ISs using DVA and PASOCS is shown below.

DVA		**PASOCS**	
A B' -> Z	(OI1)	A -> Z	(OI, Priority = 1)
A C' -> Z	(OI2)	A B C -> Z'	(NI, Priority = 2)
A B C -> Z'	(NI)		

To verify that both sets are consistent let the variables A, B, and C be set to true. In the DVA case, now only NI is matched by the environment. Thus the only setting for Z is false. In the priority case, Z is set to false by the higher priority for NI. Both sets are now consistent.

In some cases, inconsistencies are resolved by removing OI from IS. In the example above, DVA made use of the fact that NI contained variables not in OI. If however, NI is a subset or equal to an OI, the OI is simply deleted, and NI is added to IS. For example, given the instances below:

A B -> Z (OI)

A -> Z' (NI)

NI is a subset of OI, anytime OI is matched NI is also be matched, so OI can be deleted from IS, and NI can be added to IS without modification.

2.2.2 Minimization

After the operation of consistency is performed, the minimization operation can be performed. Minimization attempts to represent the same knowledge with fewer input variables, or instances as compared to the pre-minimized IS. This results in a more parsimonious representation of the knowledge base, and may enhance generalization capabilities. Where the consistency operation applies only to discordant instances, the minimization applies to concordant instances. Given below are two instances, NI is a subset OI. The IS can be minimized by removing OI.

A B C -> Z (OI)
A B -> Z (NI)

This is termed *subset minimization*. The NI is the subset, and remains in IS. If NI were a superset of some OI as shown below:

A C -> Z (OI)
A B C -> Z (NI)

then NI again remains in IS and OI is deleted. This is defined as *superset minimization*.

Now suppose there are the instances below:

A B C -> Z (OI)
A B C' -> Z (NI).

These instances state that if A and B are true, set Z to true, no matter what the value of C, and can be minimized through *one-difference equal minimization*. Given two instances with one discordant variable, if when that discordant variable is removed from both instances, the remaining instances are equal, the discordant variable can be removed from both instances and one of the remaining instances can be deleted from IS. In the example above, C is the only discordant variable and can be deleted from NI, and OI is removed from IS leaving NI as A B -> Z in IS.

Suppose that two instances, NI and OI, contain one discordant variable, and after removing that variable, NI is a subset of OI. This case is illustrated below.

A B C -> Z (OI)
A C' -> Z (NI)

NI is added to IS without modification. OI is modified to remove the discordant variable. Hence the IS would appear as

A B -> Z (OI)
A C' -> Z (NI)

after all minimizations are performed. This type of minimization is termed *one-difference subset minimization*.

The final type of minimization is *one-difference superset minimization*. Below is an example of three instances.

A B -> Z (OI1)
A' B' D -> Z (OI2)
A' B C D -> Z (NI)

In this case, NI contains one discordant variable from both OI1 and OI2. The minimization removes the discordant variable from NI. Once NI is minimized with either OI1 or OI2, it cannot be minimized with the other since the superset relationship is no longer true. For example, suppose NI is minimized with OI1, this would thus remove A' from NI, leaving NI as B C D -> Z which is no longer a superset of OI2.

The final two types of minimization find instances with only one discordant variable, remove this variable from consideration, find which remaining instance is the superset, then permanently remove the discordant instance from the superset instance. Any variety of minimization operations cannot introduce any inconsistencies to IS, since the same is represented in a more parsimonious fashion. Below is an example of IS modifications with brief explanations.

2.2.3 Instance Set Example

Examples of both DVA and Priority methods are now demonstrated. IS (Priority) shows the IS generated using PASOCS to resolve contradictions. Begin by adding the instance A B D' -> Z to the empty sets.

A B D' -> Z (NI) | IS
∅

IS (Priority)
∅

Because there are no instances in IS, NI is directly added to IS (shown as OI). In the case of priority, the instance is assigned the base priority. Another NI is presented to the systems.

A C D E -> Z' (NI) | IS
A B D' -> Z (OI)

IS (Priority)
A B D' -> Z (OI, Priority = 1)

The two instances are discordant so consistency is checked. The variable D is a discriminant variable for these instances thus there are no inconsistencies and because there are no instances concordant to NI, minimization need not be checked. NI is added to IS and IS (Priority).

The instance A B D -> Z is now presented.

A B D -> Z (NI) | IS
A B D' -> Z (OI1)
A C D E -> Z' (OI2)

IS (Priority)
A B D' -> Z (OI1, Priority = 1)
A C D E -> Z' (OI2, Priority = 1)

OI2 contradicts NI so consistency is checked first. Observe that NI and OI2 are not discriminated. To perform DVA, first select a variable in NI which is not in OI2, negate it, and add it to OI2. The variable B is the only choice, therefore B' is added to OI giving the IS below. In PASOCS the instance is added with a priority one higher than the priority of the highest contradicting instance. Therefore NI is added to IS with a priority of 2. Another NI is presented.

A B' D' -> Z (NI) | IS
A B D' -> Z (OI1)
A B' C D E -> Z' (OI2)

IS (Priority)
A B D' -> Z (OI1, Priority = 1)
A C D E -> Z' (OI2, Priority = 1)
A B D -> Z (OI3, Priority = 2)

Using DVA consistency is first checked. The variable D discriminates NI from OI2, the only discordant instance. One difference equal minimization can be performed by deleting B from NI and removing OI1 from IS. Using PASOCS, it first finds that all discordant instances are properly discriminated. Now PASOCS can check concordant instances in IS for potential minimization. PASOCS finds NI and OI1 have one discordant variable, B. Examination of the two instances less this variable, shows that they are equal. One-difference equal minimization is therefore applied. This is done by deleting OI from IS, deleting the variable D from NI, and adding NI to IS. The new ISs are shown with another NI.

	IS	
A C' D' -> Z' (NI)	A B' C D E -> Z'	(OI2)
	A D' -> Z	(OI3)

IS (Priority)	
A C D E -> Z'	(OI2, Priority = 1)
A D' -> Z	(OI3, Priority = 2)

First consistency is checked, discovering that NI and OI3 are inconsistent. A discriminant variable is found by considering the variables in NI but not in OI3. There is one, C', which is negated and added to a copy of OI3 creating the instance A C D' -> Z which replaces OI3 in IS. PASOCS is simpler. NI can be added to IS with a higher priority than any contradicting instance. Because OI3 has a priority of 2 which is the contradicting instance with the highest priority, NI is added to IS with a priority of 3. Now the instances which are concordant with NI are examined for possible minimizations. Only OI is concordant with NI. Because there are two discordant variables, minimization cannot occur and NI is added to IS. The resultant sets are shown with a final NI.

	IS	
A B' E' -> Z' (NI)	A B' C D E -> Z'	(OI)
	A C D' -> Z	(OI1)
	A C' D' -> Z'	(OI2)

IS (Priority)	
A C D E -> Z'	(OI, Priority = 1)
A D' -> Z	(OI3, Priority = 2)
A C' D' -> Z'	(OI4, Priority = 3)

Consistency is first checked. NI contradicts OI1. Discriminant variables are found by considering the variables in NI which are not in OI1. There are two, B' and E'. Each discriminant variable is negated and added to a copy of OI1 creating two instances A B C D' -> Z and A C D' E -> Z which replace OI1 in the IS. In PASOCS the contradiction between NI and OI3 is resolved by adding NI to IS with a priority of 3. Each IS is now examined for possible minimizations. Only one discordant variable exists between OI (in the DVA created IS) and NI. Again one-difference minimization is applied by considering NI and OI without the discordant variable. Since NI is a subset of OI, one-difference subset minimization does not modify NI, but does delete the variable E from OI. NI is then added to IS. The final ISs are shown below.

IS	IS (Priority)	
A B' C D -> Z'	A C D E -> Z'	(Priority = 1)
A B C D' -> Z	A D' -> Z	(Priority = 2)
A C D' E -> Z	A C' D' -> Z'	(Priority = 3)
A C' D' -> Z'	A B' E' -> Z'	(Priority = 2)
A B' E' -> Z'		

2.3 Other Instance Set Issues

It is important to note that consistency must be performed to insure proper behavior of the system. Minimization, however, is not required and is used only as an optimization. Consistency is performed first, and then minimization. Finally recall in the example on new instance one-difference superset minimization, the NI could be minimized with one of two instances, but not both, because of this, minimization is not unique.

In this work, only one output variable was used for ease of illustration. As previously stated ASOCS supports multiple output variables. This is done by considering each output variable separate from all others. Consistency and minimization are considered for each specific output variable.

2.4 Extension of ASOCS to Multi-State Variables

So far ASOCS has been discussed with Boolean variables. ASOCS can be extended to allow variables to have a number of discrete values or states. A *multi-state* (MS) variable is a variable capable of representing any finite number of states. Below are some examples of multi-state variables and the possible states for each variable.

(1) A:{1, 2, 3, 4}
(2) T:{Hot, Cold, Very Hot, Mild}
(3) C:{Red, Blue, Yellow}
(4) B:{True, False}

In these examples, the feature:value notation from chapter 1 is used. The variables A, T, C, and B are MS variables. The value of the variable is taken from the set of possible values. The current value of a variable is specified by the variable, followed by a colon, and then the element from the set of possible values. Below are some examples of this notation.

(1) A:2
(2) T:Very Hot
(3) C:Blue
(4) B:True (an alternate notation is B:1)

In the first example, MS variable A has the value 2 from the set {1, 2, 3, 4}. Likewise MS variable T has the value Very Hot, and C has the value Blue. In the final example, (4), the variable B has the value True. As in the Boolean case, MS variables are mutually exclusive. Variable A cannot have the value of 2 **and** 3, just as variable B cannot be both true and false.

An instance then, uses MS variables as input, output, or both. Below are some examples of instances with MS variables.

(1) A:2 B:1 C:Blue -> Z:1
(2) A:(4 | 6) B:0 -> Z:0
(3) T:Cold C:∅ -> Z:4

In instance (1) ASOCS sets the variable Z to true, when A has a value of 2, B is set to true, and C has a value of Blue. Instance (2) illustrates the “(|)” notation, where the "|" is the ***or*** operation common to set theory. For instance (2) the variable Z is set to false when variable A is 4 *or* 6 and B is false. Instance (3) introduces the notion of *negating* a multi-state variable. This is a separate state defined as the variable C has no value from the set of possible values. This is different from instance (2) in which a value for variable C is not known. The use of negating a multi-state variable is illustrated by an example. Suppose a station on an assemble line has the assigned task of painting various parts of fire engines. A rule exists which states, "If the fire engine is not painted, paint it red". This is expressed easily as C:∅ -> C:Red. Thus, the negation notation adds the state "none of the above" to the set of possible states for any given variable.

2.4.1 Consistency Operation with MS Variables

The consistency operation for MS variables is similar to that in traditional ASOCS. Inconsistencies arise when two or more instances in IS are not properly discriminated. As in traditional ASOCS, the addition of a discriminant variable resolves the inconsistency. The following example is used to illustrate DVA for MS variables.

A:2 C:Blue -> Z:1 (OI)
A:2 B:1 C:Blue -> Z:0 (NI)

The IS is made consistent by adding the variable B:0 to the OI. This is appropriate because ASOCS knows that B is a Boolean variable. This illustrates the similarities in the techniques between traditional ASOCS and ASOCS with MS variables. Difficulties arise when other MS variables must be added to resolve the contradiction. This is shown in the example below.

A:2 -> Z:1 (OI)
A:2 C:Blue -> Z:0 (NI)

Intuition dictates altering the OI to state A:2 C:Blue' -> Z:0. This is interpreted "set the MS variable C to any value (including ∅) except Blue". In the execution phase of the ASOCS, this is expanded to C:Red | Yellow | Green | etc. This gives the desired result, but is expensive in terms of storage. ASOCS does not restrict the number of states for a multi-state variable, therefore it must accommodate a large number. Further, the number of possible states for a given multi-state variable may not be known *a priori*, as the training set may have additional instances added at any point of the ASOCS existence. Therefore the traditional ASOCS approach of DVA is expensive and not practical. PASOCS however, does facilitate the use of multi-state variables as is shown below.

A:2 -> Z:1 (OI, Priority = x)
A:2 C:Blue -> Z:0 (NI)

With PASOCS, the NI is added to IS with priority greater than x. This is the same approach to resolve inconsistencies as used with traditional Boolean variables. Because of the simplicity of PASOCS in this regard, and the need for multi-state capabilities, this work uses PASOCS henceforth.

2.4.2 Minimization Operation with MS Variables

The minimization operation for MS variables is implemented differently than the traditional minimization approach. Given the two instances below

A:1 B:1 C:Red -> Z:1
A:2 B:1 C:Red -> Z:1

a more parsimonious representation can be found. The two instances are minimized to the instance shown below.

A:(1 | 2) B:1 C:Red -> Z:1

Therefore, when the variable A is set to A:1 **or** A:2, this single instance may be matched.

Below three instances are given.

A:1 B:1 C:Red -> Z:1 (OI1)
A:2 B:1 C:Blue -> Z:1 (OI2)
A:1 B:1 C:Blue -> Z:0 (OI3)

It is tempting to create a single erroneous instance

A:(1 | 2) B:1 C:(Red | Blue) -> Z:1.

To see the error in this instance, consider the case of variable A set to 1, variable B set to 1, and variable C set to blue. The valid setting of Z is now unknown due to the erroneous instance and the OI3.

Minimization therefore is only applied to logically combine states of a variable when all other states are equal. This is best illustrated with the examples below.

<u>Example 1</u>

A:1 B:0 C:Red -> Z:0 (OI1)
A: (2 | 3) B:0 C:Red -> Z:0 (OI2)
A: (1 | 2 | 3) B:0 C:Red -> Z:0 (OI1 minimized with OI2)

Example 2
A:1 B:0 C:Red -> Z:0 (OI1)
A:2 B:0 C:Red -> Z:0 (OI2)
A:1 B:1 C:Red -> Z:0 (OI3)
A:1 B: (0 | 1) C:Red -> Z:0 (OI1 minimized with OI2)

This instance cannot be further minimized because the values for the variable B in the two instances are not the same. Consider one final example to show how minimization may cause further minimizations.

Example 3
A:1 B:0 C:Red -> Z:0 (OI1)
A:2 B:0 C:Red -> Z:0 (OI2)
A:(1 | 2) B:0 C:Blue -> Z:0 (OI3)
A:(1 | 2) B:0 C:Red -> Z:0 (OI4)
A:(1 | 2) B:0 C:(Red | Blue) -> Z:0 (OI3 further minimized with OI4).

Minimization is not necessary for proper functioning, but is desirable since it assists in generalization and parsimony.

2.5 PASOCS System Overview

This section, gives a very brief overview of a PASOCS. PASOCS is extended in section 5.3 to support continuous input features. A much more detailed description is given in [12]. Figure 1 shows a functional diagram of a PASOCS system.

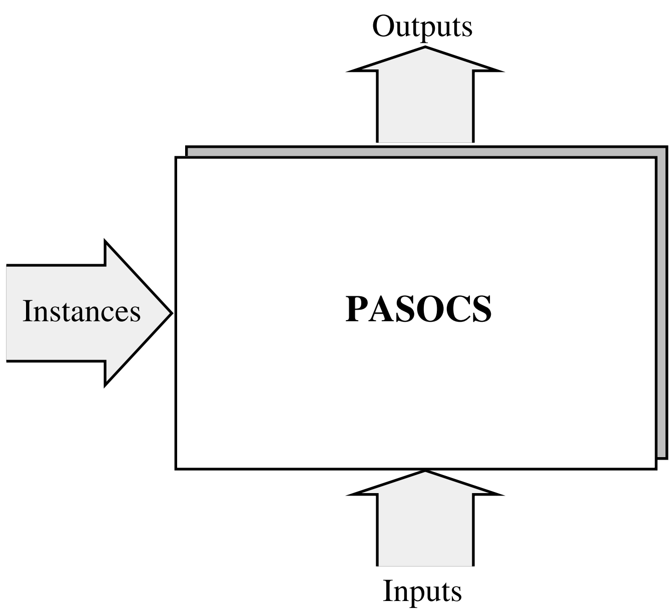


Figure 2.1 A functional representation of an PASOCS system

Figure 2.1 shows that PASOCS maps inputs to outputs, through instances input to the system during the training phase. A PASOCS is shown in figure 2.2. The adaption unit is used in training and is connected to the PASOCS logic network by a communication path. During the execution phase, the adaption unit performs no function. The PASOCS logic network is organized as a binary tree.

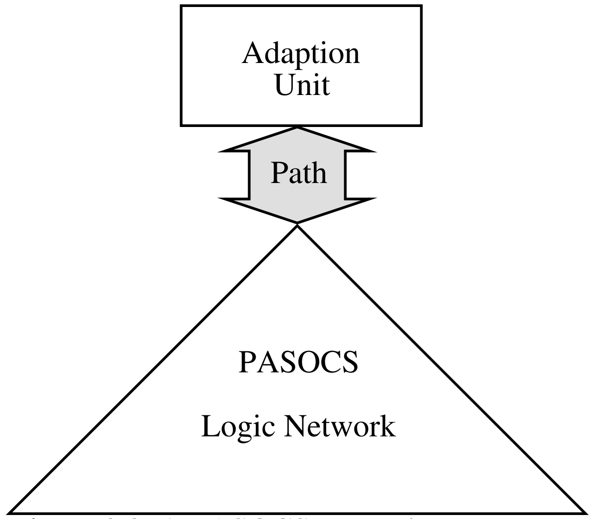


Figure 2.2 A PASOCS network

The general structure of each node of the tree is shown in figure 2.3. Each node is capable of storing an instance, the output and priority of the instance and a number of flags used in learning. For example, the flag *instance_matched* is set if the instance stored at the node contradicts, and is not discriminated from NI.

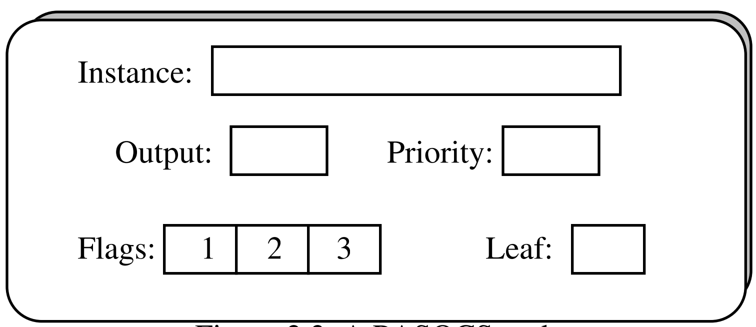


Figure 2.3 A PASOCS node

Figure 2.4 shows the algorithms broadcast and gather. The broadcast algorithm is used to pass a message down the tree from the root to the leaves of the tree. The gather algorithm is used to collect information starting at the leaves of the tree, up to the root of the tree. The *test* parameter to the broadcast function states a test for each node of the tree to perform. For example, the test “match” is performed to determine if an instance is matched by the PASOCS logic network. Any node whose stored instance matches the instance passed as a parameter to broadcast, sets a flag to true. The parameter *test_at_node* to the gather function performs a test such as “output highest priority” which is used to set the output to the output value of the matching instance with the highest priority. Figure 2.5 shows the test “match” and figure 2.6 shows the test “output highest priority”.

```
broadcast(instance, test)
  From the root, pass the message contained as parameters to each
      child.
  Continue for each subtree until the message has reached
      the leaves of the tree.
  end broadcast

gather(test_at_node)
  From the leaves of the tree, each node passes the information
      requested as the parameter to gather.
 end  gather
```

Figure 2.4 The algorithms broadcast and gather

```
match(instance)
  node.instance_matched:= True;
  for each feature in instance  {
      if (instance.feature.state ≠ Don't_Care)  {
          if  (PASOCS_instance.feature.state ≠ Don't_Care)  {
              if (instance.feature.state ≠
                              PASOCS_instance.feature.state)  {
                  node.instance_matched:= False;
                  }  // end if states ≠
              }  // end if PASOCS = Don't_Care
          }  // end if instance = Don't_Care
      }  // end for each discrete feature in instance
 end  match
```

Figure 2.5 The test match

```
output_highest_priority                    // non-leaf node
  if (node.matched instance = True)  {     // Consider this nodes output
      if (node.priority > left_child_priority) AND
              (node.priority > right_child_priority)
          pass_to_parent(node.output, node.priority);
      else if (left_child_priority > right_child_priority) AND
              (left_child_priority > node.priority)
          pass_to_parent(left_child_output, left_child_priority);
      else if (right_child_priority > left_child_priority) AND
              (right_child_priority > node_child_priority)
          pass_to_parent(right_child_priority, right_child_output);
      else if (right_child_priority > left_child_priority)
          pass_to_parent(right_child_priority, right_child_output);
      else
          pass_to_parent(left_child_output, left_child_priority);
      }  // end if node.mathced.instance = True
  else  {              // node.matched.instance = False
      if (right_child_priority > left_child_priority)
          pass_to_parent(right_child_priority, right_child_output);
      else
          pass_to_parent(left_child_output, left_child_priority);
      }  // end else node.matched.instance = False
 end  output_highest_priority

pass_to_parent()
  Forward the parameters to the single parent node.
 end  pass_to_parent
```

Figure 2.6 The test output_highest_priority

Figure 2.7 shows the algorithm to find the output of the PASOCS network for a given instance. This algorithm is the PASOCS execution algorithm and is also used in PASOCS training.

```
generate_PASOCS_output(instance)
  broadcast(instance, match);
  output:= gather(output highest priority);
  return (output);
 end  generate_output
```

Figure 2.7 The algorithm generate_PASOCS_output

Figure 2.8 shows the algorithms for the training phase of PASOCS. The training instance is broadcast to the PASOCS network with the test “not discriminated and discordant.” Each node performs the test and sets the appropriate flags. The highest conflicting priority is gathered and the instance is added to the PASOCS network with priority one greater than the highest conflicting priority. Prior to adding NI, minimization is performed as discussed in sections 2.2.2 and 2.4.2.

PASOCS_learn(*instance*)
broadcast(*instance*, not discriminated and discordant);
priority:= gather(largest conflicting priority);
minimization(*instance*);
add_instance(*priority*+1, *instance*);
end PASOCS_learn

not_discriminated_and_discordant // test for setting flags
Determine if instance is discordant as shown thusfar.
Determine if NI is discriminated from instance at node.
end not_discriminated_and_discordant

add_instance(*priority*, *instance*)
Add *instance* to the PASOCS instance set.
Allocate a free node in the network.
node.instance_priority:= *priority*;
node.instance_output:= *training_instance.output*;
node.instance:= *instance*;
end add_instance

largest contradicting priority // test at node for gather
if (*node.instance_contradicted* = True)
pass_to_parent(max(*node.priority, left_child.priority, right_child.priority*));
else
pass_to_parent(max(*left_child.priority, right_child.priority*));
end largest contradicting priority

Figure 2.8 PASOCS training algorithms

2.6 Analysis of PASOCS

PASOCS, like ASOCS, gives precedence to the most recent instance entered. This is ideal if there are no errors or *noise* in the data set the PASOCS is using for training. If the last instance the PASOCS sees contains an error, PASOCS learns the error. Typically, the data set is not guaranteed to be error-free.

PASOCS is guaranteed to learn any valid IS without *iteration*. Iteration is used to present each instance of the training set to the learning system a number of times.

Basic PASOCS does not generalize to well to instances it has not seen. PASOCS sets the output variable only when an instance is matched by the environment. If the instance is does not match the environment exactly, the output is not set by PASOCS. There are some extensions which allow PASOCS to generalize [11][17].

When the logic network is organized as a binary tree, the complexity of learning each instance and execution in PASOCS is $O(2\log_2 n)$ where n is the number of instances in the PASOCS IS.

Chapter 3

An Introduction to ID3

This chapter presents a supervised learning system from the machine learning paradigm. ID3 is a machine learning system which builds decision trees inductively [14]. The resulting trees then are capable of classifying based upon decisions determined by the input features. Because of the utility of the these trees, ID3 is demonstrated as a SL for the learning methodology presented in chapter 5.

This chapter begins with an example decision tree based on a training set. Then the algorithm for building the tree used in the example is presented. The chapter concludes with an analysis of ID3.

3.1 Introductory example

The training set in figure 3.1 is taken from [14]. The decision is whether to play golf based on current weather conditions. The input features are temperature, humidity, wind, and outlook. Unlike the similar data set given in chapter 1, this data set has no continuous values. ID3 supports only discrete features, therefore temperature and humidity have previously been coded by some preprocessing. There are variations to ID3 which support continuous values[15]. This work shows another alternative. In 9 cases the decision is positive; in 5 cases the decision is negative (14 cases total).

No	Features				Class
	Outlook	Temp	Humidity	Windy	
1	sunny	hot	high	false	N
2	sunny	hot	high	true	N
3	overcast	hot	high	false	P
4	rain	mild	high	false	P
5	rain	cool	normal	false	P
6	rain	cool	normal	true	N
7	overcast	cool	normal	true	P
8	sunny	mild	high	false	N
9	sunny	cool	normal	false	P
10	rain	mild	normal	false	P
11	sunny	mild	normal	true	P
12	overcast	mild	high	true	P
13	overcast	hot	normal	false	P
14	rain	mild	high	true	N

Figure 3.1 The golf training set

Figure 3.2 shows the decision tree produced by applying the ID3 algorithm to the golf training set. Input features appear in the nodes of the tree, with values for the features representing branches of the tree. The leaves of the tree are results or output. To arrive at an output for a given input instance, ID3 simply starts at the root, and begins testing. A branch of the tree is selected when the value for the input feature of the given instance is equal to the value of the feature associated with that branch.

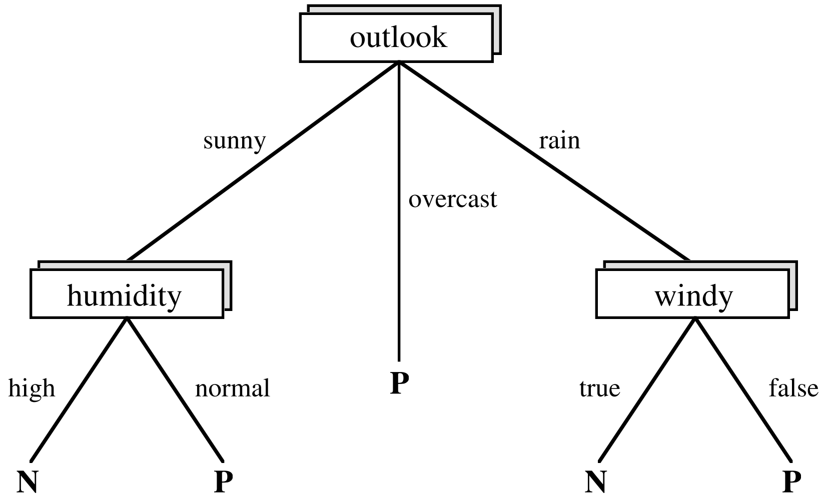


Figure 3.2 The decision tree produced from the golf training set

For example, a particular Saturday morning may be described as

Outlook: sunny
Temperature: cool
Humidity: high
Windy: false

A decision based on these values begins at the root. The value of the outlook feature is tested. The branch labeled "sunny" is selected, which places the decision at the node labeled humidity. The value of the humidity feature is tested. The branch labeled "high" is selected, to arrive at the leaf decision of "N". Thus "N" is the output.

3.2 The ID3 Learning Algorithm

The ID3 learning algorithm is shown in figure 3.3. The algorithm begins by selecting a *training window* from the training set. A training window is just a subset of the training set, but may contain the entire training set. In creating the tree shown above in figure 3.2, the entire training set of figure 3.1 is used as the training window. ID3 uses the training window to gather information to build the tree.

ID3 Algorithm

1. Create a training window.

2. Generate tree based on training window by:
   a) Apply selection criterion to find feature for current node
   b) Create subset training window.
   c) Recursively build each subtree

3. Apply instances outside training window to decision tree.
   IF: (No misclassified instances)
      Stop.
   else
      add misclassified instances to training window,
      goto step 2.

Figure 3.3 The ID3 learning algorithm

A *selection criterion* is applied to the training window. A selection criterion is a metric used to determine which feature is best at classifying the instances of the given training window. This metric is used to determine which feature should become the conditional test at that node. In the tree shown above in figure 3.2, the outlook feature is deemed "best" at classifying the training window. The selection criterion for ID3 is found by applying a series of information based equations using information obtained from the training window. Equation 3.1 is applied first to determine the information required to classify the training window.

$$I(o_1,\ldots,o_m) = \sum_{j=0}^{m} \left(-\frac{o_j}{\sum_{j=0}^{m} o_j} \log_2\left(\frac{o_j}{\sum_{j=0}^{m} o_j}\right)\right), \qquad \text{(eq. 3.1)}$$

where o_j is the cardinality of the j^{th} output class and m is the number of possible output classes. For the golf data set, only two output classes exist, and equation 3.1 reduces to equation 3.2.

$$I(p,n) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}, \qquad \text{(eq. 3.2)}$$

where p is the number of positive instances in the training window, and n is the number of the negative instances in the training window. Because the entire training set is used as the training window for the tree of figure 3.2, $p = 9$, $n = 5$, and $I(p,n)$ is therefore 0.940. Note that $I(p, n)$ does not depend on the values of the input features, but depends only on information acquired from the training window. Thus $I(p,n)$ is constant for all features using the same training window.

The expected information for the tree with a given feature at the root is then computed by equation 3.3.

$$E(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p+n} I(p_i, n_i), \qquad \text{(eq. 3.3)}$$

where A is a given feature (attribute), i is the i^{th} possible value for feature A, p_i is the number of positive instances with the i^{th} value for feature A, and n_i is the number of negative instances with the i^{th} value for feature A. The example below shows the use of equation 3.3 with the outlook feature. In this example, five of the 14 instances of figure 3.1 have the first value ("sunny") for feature outlook. Of these five instances, two are positive and three are negative. Continuing, other values are

$$p_1 = 2 \quad n_1 = 3 \quad I(p_1, n_1) = 0.971$$
$$p_2 = 4 \quad n_2 = 0 \quad I(p_2, n_2) = 0$$
$$p_3 = 3 \quad n_3 = 2 \quad I(p_3, n_3) = 0.971$$

$$E(\text{outlook}) = \frac{5}{14} I(p_1, n_1) + \frac{4}{14} I(p_2, n_2) + \frac{5}{14} I(p_3, n_3) = 0.694 .$$

Equation 3.4 shows how the information required and information expected are used together to determine the best feature for a particular node of a tree.

$$gain(A) = I(p, n) - E(A), \qquad \text{(eq. 3.4)}$$

where A is a feature, $I(p,n)$ is the information required for the feature A (computed from eq. 3.2 above), and $E(A)$ is the information expected by branching on feature A (computed by eq. 3.3 above). The feature selected as the conditional for a particular node is the feature which maximizes the *gain*. Because $I(p,n)$ is constant for all features, maximizing gain is the same as minimizing $E(A)$. The value of the *gain* function is shown below for all features.

$$gain(\text{outlook}) = 0.246$$
$$gain(\text{temperature}) = 0.029$$
$$gain(\text{humidity}) = 0.151$$
$$gain(\text{windy}) = 0.048$$

Based on these results, the feature outlook is selected as the root conditional. The algorithm inserts the node representing outlook, and generates a branch for each of the possible values of the feature outlook. Because outlook has three possible feature values, “sunny”, “cloudy”, and “rain”, there are three branches from the node for the outlook feature.

Each subtree then is built recursively by the algorithm above, using a subset of the original training window. Each subset training window is created by considering the training window instances which have the same value for the feature of the parent node. Using the training set above, three subset training windows are created. All instances which have the value “sunny” for the feature outlook, are placed in one subset training window. Therefore the subset training window denoted as T_{sunny} has five instances, as shown in figure 3.4. The number column in figure 3.4 shows the numbers of the original training set. The feature which is used to partition the training window is removed from the instances. The subset window, T_{sunny} is used to create the subtree connected to the root by the branch with the value “sunny”. It is easy to see from figure 3.4 why the feature “humidity” is selected as the feature for the node connected to the root.

No	Features			Class
	Temp	Humidity	Windy	
1	hot	high	false	N
2	hot	high	true	N
8	mild	high	false	N
9	cool	normal	false	P
11	mild	normal	true	P

Figure 3.4 The subset training window T_{sunny}

After an ID3 tree has been created based on a training window, the tree is used to classify the instances which are not in the training window (if the training window ≠ training set). If all instances outside the training window are properly classified, the algorithm stops. If however, there exists an instance which is misclassified, that instance may be added to the original training window, and a new tree may be created based on this new training window.

3.3 Analysis of ID3

Selection of the window size can cause some potential problems. If the window is too small, several instances are added after the first tree has been created. A new tree is now created using the new training window. However, the new tree is not guaranteed to classify the instances correctly outside the training window which were previously correctly classified. Thus, the training window can slowly grow to encompass the entire training set. Therefore, it may be simpler to begin with the entire training set thus creating the tree only once. Current versions of ID3 and variations to ID3 do not use a training window. [15]

The time complexity of execution is bounded by $O(m)$, where m is the number of input features. Each input feature may occur at most once down each path of the tree.

The time complexity of learning considers each training instance once to compute the statistics used in the gain function. This has time complexity of $O(n)$, where n is the number of instances in the training set. Each feature is considered as the test at each node. This consideration is performed for each of the m levels of the tree giving a final time complexity of $O(m^2n)$.

Chapter 4

The Competitive Classifier

The previous chapters have described two supervised learning systems from the machine learning paradigm which accept only discrete valued inputs. Both systems have promising learning capabilities. This chapter presents the *Competitive Classifier* (CC) which is neural network model whose purpose is to produce discrete codings from continuous valued inputs. Recall from chapter 1 that the CC is connected to a supervised learner (SL). Information is fed forward from the CC to the SL. Part of this information is the discrete codings for continuous valued input features. The SL uses this information to attempt to solve a problem. During the training phase, the SL generates feedback for the CC to attempt to improve the codings of the continuous valued input features. This chapter discusses what information the CC produces and how the CC produces this information. The chapter also discusses the feedback accepted by the CC and how this feedback is used by the CC to potentially improve codings. How the feedback is produced is the subject of chapter 5.

The chapter begins with the motives of the CC, without much detail on how they are accomplished. Next the structure of the CC and details of its functions are addressed. The chapter concludes discussing the idea of confidence metrics and how the CC uses confidence to support its functionality.

4.1 Introduction to Classification Through the CC

Chapter 1 shows the need for coding of continuous valued inputs. This coding could be accomplished with a preprocessor coarse-coding technique [18][7]. However, the coarse-coded codings are not guaranteed to be useful for solving a given problem. Figure 4.1 shows a continuous valued feature with a potential coarse-coded codings.

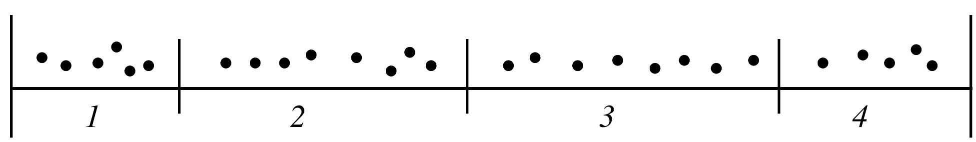


Figure 4.1 Coarse-coded codings

Given coarse-coded codings, the codings could be improved by using information available from the problem. Possible improvements may include creating a new coding for the feature, or changing coding boundaries. Figure 4.2 shows the same feature as figure 4.1, but with potentially improved codings. A coding has been created, and values previously coded in coding 3 are now coded in coding 4 due to a change in the boundary between the two codings.

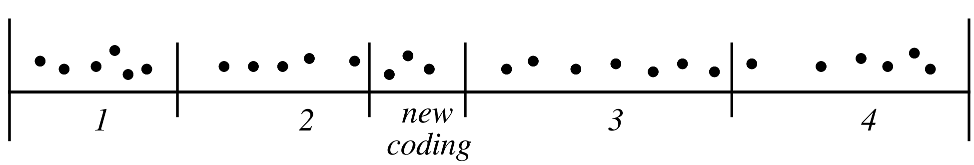


Figure 4.2 Potentially improved codings

The CC improves codings based on feedback from a SL. Based on feedback, the CC may perform any one of three functions:

1) Reinforce current codings.
2) Adjust the boundary between two codings.
3) Split a coding to create a new coding.

To understand how these functions are performed, the structure of the CC is presented in the next section. After the structure is presented, the details of how these functions are carried out are given.

4.2 The Structure of the CC

The CC is a collection of *feature coders (FC).* Each FC is a single layer neural network model derived from the competitive learning model [16]. Single layer means that the CC has two layers of *nodes* connected by a single layer of adjustable *weights*. A node is a neurally inspired unit capable of accepting inputs and generating an output. A weight is a real value on the interval of [0, 1]. Figure 4.3 gives a diagram of a FC. Figure 4.4 shows a CC made up of three FCs and thus capable of coding three continuous features.

4.2.1 Input Nodes

The input layer of a FC consists of an input node. The node represents one continuous valued feature. The output of an input node is a real valued number on the interval [0, 1]. All continuous valued inputs are normalized. The need for normalizing inputs is discussed later. Each input node is connected to one or more output nodes by an adjustable weight. A weight is also referred to as a prototype. The prototype associated with the connection from input node i, to output node j, is abbreviated w_{ij}, as shown in figure 4.3.

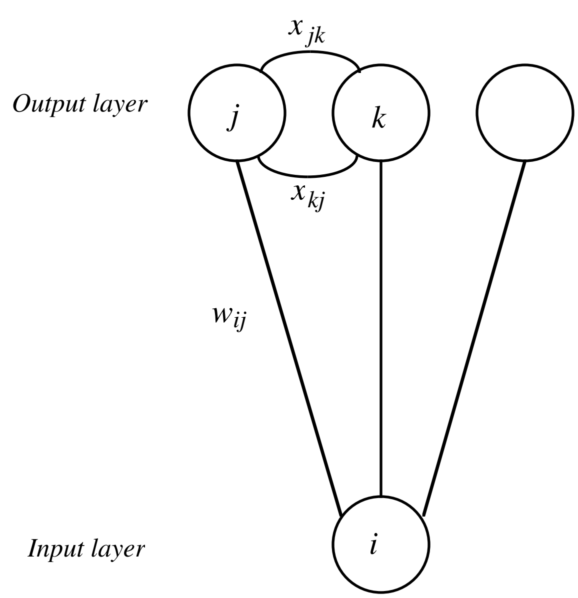


Figure 4.3 A FC

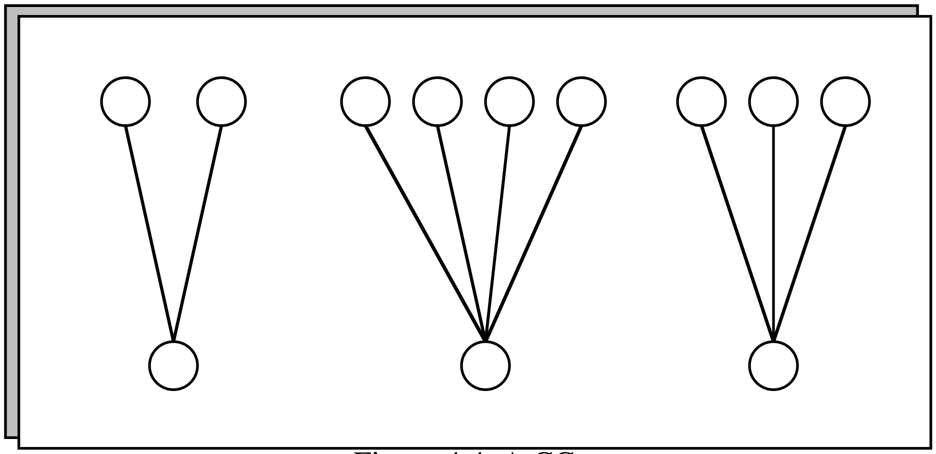


Figure 4.4 A CC

4.2.2 Output Nodes

The output layer of the CC consists of a collection of output nodes. *An output node represents one discrete coding for a continuous feature*. Each output node receives input from one input node. The nodes which receive input from the same input node are grouped together to form a *cluster*. A cluster is a group of output nodes and represents the possible codings for one continuous valued feature. Within a cluster, every output node is connected to every other output node. This is not shown in figure 4.3 where only nodes j, and k are connected. Associated with these intra-cluster connections are real valued *inhibitory weights*. An inhibitory weight is simply a weight which is typically negative. The inhibitory weight from output node j, to output node k is denoted x_{jk}, as shown in figure 4.3. Previous competition based learning models, have symmetric inhibitory weights [3][5][6][16]. Inhibitory weights of a FC are not symmetric, and are dynamic, changing over time. Each output node has an output value o_j.

The output of each node is found by a two step process. Step one computes

$$a_j = \frac{1}{(w_{ij}-m_i)^2}, \qquad \text{(eq. 4.1)}$$

where a_j is the activation of output node j, w_{ij} is the weight from input node i to output node j, and m_i is the output of input node i (the continuous value of the feature). The input m_i is normalized and squared to force the denominator of equation 4.1 to be on the interval [0, 1]. The activation is then used to determine the final output of the node by the rule given in equation 4.2.

$$o_j = a_j - \sum_{\substack{k=1,\\ k\neq j}}^{n} (a_k * x_{kj}), \qquad \text{(eq. 4.2)}$$

where n is the number of output nodes in the cluster for the given feature. In this equation, the initial activation of a node is decreased by the sum of the activations weighted by the inhibition of all other nodes in the output cluster. The inhibition between nodes causes a competition. The output node nearest the prototype has a better chance of winning the competition. If the input falls between two prototypes, the inhibition determines which node has the highest output value which is the *winning* node. By changing the inhibition between two nodes, the winning node for a given input may change. The inhibitory weights between two nodes determines the boundary the two codings the nodes represent.

4.2.3 The CC Control Unit and Output from the CC

The CC has a control unit which coordinates the information fed forward to the SL. The control unit has the ability to determine which output node has the highest output value, and also the output node with the next highest output value, for a given input feature. The possible codings are determined by the winning node and the second highest node and are forwarded to the SL, along with a *confidence metric* of the feature. A confidence metric is a real valued number which is typically derived from the final outputs of the winning nodes and is discussed in section 4.4. While the control unit does not give a name to the codings it produces, for clarity the examples presented in this chapter and the next use meaningful names for the codings. As an example of what the CC forwards, suppose that our problem has three continuous valued features (named Temperature, Color, and Height) which the CC is to code. If the CC codes the feature Temperature as first "hot", second "mild" and confidence 0.8, the stream {Temperature:hot, mild (0.8)} is forwarded to the SL for the feature Temperature. The confidence of each feature is shown in parenthesis, after the discrete coding names. An entire stream of information forwarded from the CC to the SL might be as in example 4.1.

{Temperature:hot , mild (0.8);
Color:blue, green (0.95);
Height:tall, very tall (0.75)}

Example 4.1 An information stream forwarded from the CC to the SL

The SL uses the codings information forwarded from the CC and combines it with other (previously) discrete data to learn a problem. Based upon results discussed in the next chapter, the SL generates feedback which is sent back to the control unit of the CC. The control unit then uses this information to produce potentially better codings by adjusting the CC.

4.3 CC Function Implementations

The functions of the CC are

1) Reinforce current codings.
2) Adjust boundary between two codings.
3) Split a coding to create a new coding.

4.3.1 Reinforcing Current Codings

Current codings in the CC are reinforced when the codings are deemed "good." The SL deems the codings as good, when they lead the SL to a correct output as discussed in chapter 5. Codings are reinforced by adjusting the prototype as in equation 4.3.

$$w_{ij} = w_{ij} + \frac{w_{ij} - m_i}{r_j} \quad , \qquad \text{(eq. 4.3)}$$

where w_{ij} is the j^{th} prototype, m_i is the value of the i^{th} input node (feature), and r_j is the number of times the j^{th} output node has been reinforced. The value r_j is incremented each time the j^{th} coding is reinforced. This simple rule approximates the average of all inputs coded in the j^{th} output coding. As r_j becomes large, the changes to $w_{i,j}$ become small.

4.3.2 Adjust Boundary Between two Codings

Recall that the two output nodes with the highest output values are winning codings, and both codings are forwarded to the SL. Two codings are forwarded so that the SL may try a second coding for a given feature. The details of when to try a second coding and which feature should try a second coding are discussed in chapter 5. If the SL uses a second coding and the result is favorable. This information is fed back to the CC through the feedback mechanism. If the input value is close to a boundary between two codings, the output of the nodes representing those two codings is approximately equal. By attempting both codings for a given feature when necessary, the SL may find cases where a useful coding has the second highest output. Useful codings should have the highest output value. Adjusting the inhibition between nodes changes the boundary between the codings and may increase the chance that the useful coding is selected next time the CC processes an input close to that value.

Equations 4.4 and 4.5 show how inhibitory weights are adjusted. In each equation, node *k* is the node with the highest output value, node *j* represents the node associated to the coding which the SL used to correctly solve the problem (the node with the second highest output value).

$$x_{jk} = x_{jk} - (1 - \frac{o_j}{o_k}) \text{ and} \qquad \text{(eq. 4.4)}$$

$$x_{kj} = x_{kj} + (1 - \frac{o_j}{o_k}), \qquad \text{(eq. 4.5)}$$

where x_{jk} is the inhibitory weight from node *j* to node *k*, and o_j and o_k are the outputs of nodes *j* and *k* respectively. The inhibition from the node representing the correct coding (node *j* from above) to the node representing the incorrect coding (node *k* from above) is increased by equation 4.4. This has the effect of decreasing the output value of the node associated with the incorrect codings. By equation 4.5, the inhibitory weight from the previously highest valued output node (node *k*) to the correctly coded node (node *j*) is decreased. This has the effect of increasing the output value of the correctly coded node (node *j*). Recall that inhibitory weights are typically negative. Figure 4.5 shows an example where only the pertinent output nodes are shown. In this example, node *k* has won the competition. However, the SL has found the coding of node *k* to be a more useful coding. Thus feedback is generated to adjust the boundary between the two nodes by equations 4.4 and 4.5. The numbers inside each node represent the final output value for the node. The numbers in parenthesis are the initial inhibitory weights. The numbers in bold and parenthesis are the inhibitory weights after adjustment by equations 4.4 and 4.5. Figure 4.5 shows a resultant difference in codings from the change in inhibition.

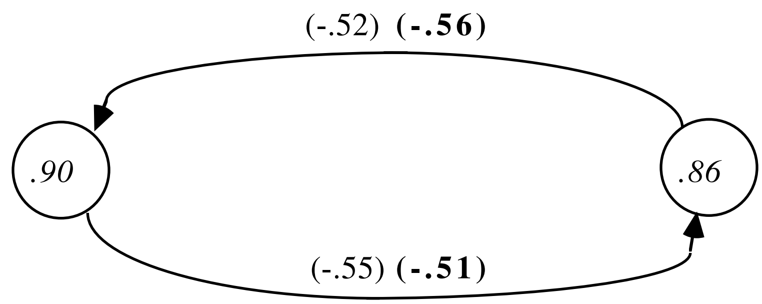


Figure 4.4 An inhibition example

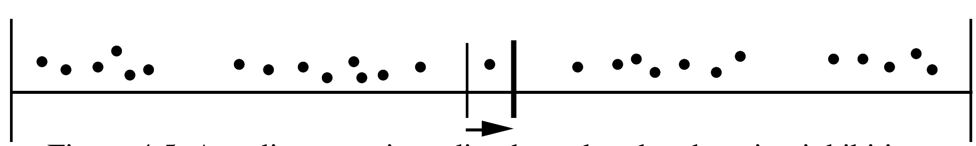


Figure 4.5 An adjustment in coding boundary by changing inhibition.

4.3.3 Split a Coding to Create a New Coding

When the CC receives feedback that the codings produced are not sufficient to assist solving the problem at hand, the CC may create a new coding for a given input feature. The input feature selected is an option partially directed by the user.

Prior to discussing which feature is selected, the process of splitting a coding to create a new coding is addressed.

The CC must allocate a new output node to create a new coding. The CC then connects this node with the appropriate input feature. As part of this connection, there is an associated prototype. The prototype is initialized to the input value which has caused the coding to split. Thus if node *j* is the new node, then $w_{ij} = m_i$. The value r_j (number of times node *j* has been reinforced in equation 4.3) is set to 1.

The inhibitory connections are also initialized at this point. The initial inhibitory weights are based on the output values of the various nodes. Therefore if a node has little response (low output value) to a given input, there is no need to inhibit that node's output. However, the node which previously miscoded the input should be inhibited. All inhibitory connections to the new node *j* are initialized by equation 4.6 shown.

$$x_{kj} = \overline{o_k} * \left(\frac{-1}{p}\right), \qquad \text{(eq. 4.6)}$$

where x_{kj} is the inhibitory weight from node *k* to new node *j*, $\overline{o_k}$ is the normalized output value of node *k*, and *p* is a user defined parameter used to set the width of the new coding as discussed below. (Empirical results typically used a value of 1.0 $\leq p \leq$ 2.0). This equation gives a value on the interval [-1.0, 0] for the inhibitory weights. This is important because the inhibitory weight adjustment rules (equations 4.4 and 4.5) simply reapportion the inhibitory weights between two nodes by adding a derived value to the initial value of the inhibitory weights. The derived value is always less than 1.0. If the initial value of an inhibitory weight is much greater than 1.0, equations 4.5 and 4.6 do not result in any significant changes to the inhibitory weights. The inhibitory connection from new node *j* to node *k* is given in equation 4.7

$$x_{jk} = -\overline{o_k}(1 + x_{kj}). \qquad \text{(eq. 4.7)}$$

The inhibitory weight from the newly created coding to the existing coding is again based on the normalized output of the existing node. However, because the inhibitory weight is based on the result of equation 4.6, which has the parameter *p*, the CC can make new codings very narrow or very broad. This is shown in figure 4.6 where a new coding is created (denoted as *j*). The prototypes for each coding are given by the "*". Two boundaries are drawn in with representative values for the parameter *p*. When $p \approx 1$, the boundary between the two codings is closer to the new coding, thus creating a narrow coding. When $p \approx 2$, the boundary shifts towards the middle of the two codings. Finally, if the output value of the node is very low, there is very little inhibition between the two nodes.

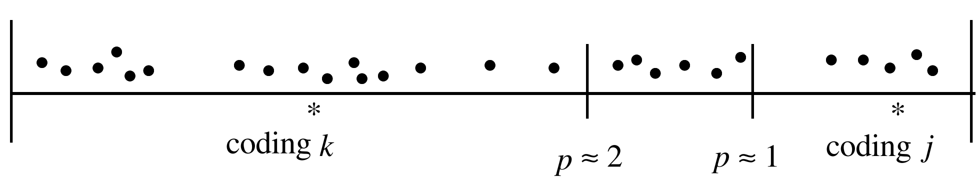


Figure 4.6 A new codings created with different boundaries as determined by *p*.

The input feature selected for creation of a new coding is typically based on the confidence metric. Calculation of the confidence metric is now discussed, followed by the selection of the feature to split.

4.4 Calculation of the Confidence Metric and its use in New Codings Creation

Much of what the SL does is dependent on the confidence metric fed forward from the CC. The confidence of the features is used to determine which alternate codings should be attempted and in what order. Typically the feature with the lowest confidence is selected as the feature which attempts an alternate coding. Refer to example 4.1 which is repeated below. When the SL receives this information, it first attempts to solve the problem with the first coding (hot, blue, and tall). If however, these codings do not assist the SL to solve the problem, it would attempt an alternate codings for a feature. The SL always selects the feature with the lowest confidence first to try an alternate coding. In this example, the input feature with the lowest confidence is the feature Height. Thus the SL would use the alternate coding of Height:very tall, and attempt to solve the problem using this coding. The various methods of calculating confidence effect how the SL attempts other alternate codings. Currently there are six different methods used to find the confidence of each coding. Each method is discussed and analyzed below.

{Temperature:hot, mild (0.8);
Color:blue, green (0.95);
Height:tall, very tall (0.75)}

Example 4.1 An information stream forwarded from the CC to the SL

4.4.1 Normalized Ratio

The normalized ratio method sums up the output values for all nodes of a given cluster. The confidence of the first coding is found by dividing the highest valued output node by this sum. The confidence of the second coding is found by dividing the second highest valued output node by this sum. Equation 4.8 shows this method.

$$c_j = \frac{o_j}{\sum_{k=1}^{n} o_k}, \qquad \text{(eq. 4.8)}$$

where c_j is the confidence of the j^{th} node, o_j is the output value of node j, and n is the number of output nodes in the particular output cluster. Node j is either the node with the highest output value, or the node with the next highest output value. For this method, the confidence of the feature is the confidence of the highest valued output node. The confidence is normalized since it is compared to the confidences of other features.

This method has the appeal that the closer an input value is to a particular prototype, the higher the confidence for that coding. Anytime the first coding has a very high confidence, the confidence of the second coding is low. When the SL considers using a second coding, the feature selected to attempt a second coding is the feature with the lowest first coding confidence. If the alternate coding of the feature with the lowest confidence gives unfavorable results, no other alternate codings are attempted. For this method the confidence calculated for the second coding is not used.

4.4.2 Normalized Difference

The normalized difference method is based on the method above. In this method, the confidence of the first and second coding is calculated as above, and then the confidence of the second coding is subtracted from the first giving the difference between the two confidences. This difference is forwarded to the SL as the confidence of the feature, along with the two codings. Equation 4.8 from above is used to find c_h (confidence of the highest valued output) and c_n (confidence of the second highest valued output). Then the confidence for feature i is

$$c_h - c_n .$$

An input value on the boundary between two codings now has a low confidence. While input values which are away from the boundary, give a high confidence. The SL uses this confidence to determine which feature should have a coding changed. Let coding 1 be the coding with the highest output value, and coding 2 be the coding with the second highest confidence for a given input feature. If the input value for this feature were near the boundary between coding 1 and coding 2, then the normalized difference confidence method gives this feature a low confidence. The SL may then replace coding 1 with coding 2. If coding 2 is found by the SL to be useful in solving the problem, a message to adjust coding boundaries is sent back to the CC. Thus the boundary between codings 1 and 2 is adjusted to make it more likely (but not guaranteed) that the given input value is coded into coding 2 in the future.

There is a subtle difference between the two methods discussed so far. The normalized ratio method gives mimimum confidence to the feature with a value farthest from the prototype of the winning coding. The normalized difference method gives minimum confidence to the feature with a value closest to the boundary between the two codings.

4.4.3 Normalized Ratio Threshold

In the two methods discussed above, only one alternate or second coding may be attempted. In this method, any coding with a confidence below a certain threshold may be attempted. The confidences of the codings are computed as in the normalized ratio method. However, instead of finding the coding with the lowest confidence, changing that coding, and attempting just one alternate instance, all codings with confidence below a certain value may be changed. In order to generate useful feedback, the SL tries second codings one feature at a time. Thus if there are four continuous valued features with normalized ratio confidences below the threshold, the SL changes the ccoding of the feature with the lowest confidence value to the second possible coding and leaves all other coding with the first codings. If this change does not assist in solving the problem (as discussed in chapter 5), this coding is changed back to the original coding and the feature with the second lowest confidence is selected and an alternate coding is attempted for that feature. The process terminates when a second coding is found to assist the SL in solving the problem, or all alternative codings have been tried. Because only one feature is allowed to change codings at a time, the time complexity of trying alternate codings $O(m)$, where m is the number of input features.

4.4.4 Normalized Difference Threshold

The normalized difference threshold method is similar to the normalized ratio threshold method discussed above. The normalized difference threshold method uses the differences in the normalized difference method discussed in section 4.4.2. As opposed to finding the input feature with the lowest normalized difference and try only one alternate coding, all input features with confidence below a certain threshold potentially have their codings changed as in the normalized ratio threshold method. Example 4.2 shows a potential information stream forwarded from the CC to the SL. This example uses the same information as in example 4.1 where the difference of the confidences has been computed and passed as the single confidence metric for the input feature. Using the normalized difference threshold method, with a threshold value of 0.5, the SL would first change the coding of the feature Height from “tall” to “very tall”. If this change produced desirable results in the SL, feedback would be generated instructing the CC to change the boundary between “tall” and “very tall”. If however, the SL continued to experience undesirable results, the Height feature would be changed back to “tall” (the first coding), and the Temperature feature would be changed to “mild”. Again if this produced desirable results in the SL, feedback is generated causing the CC to adjust the boundary between "hot" and "mild". If changing the Temperature feature produces the undesirable results in SL, then feedback to create a new coding is generated and passed back to the CC. Because the Color feature has a confidence above the threshold of 0.5, an alternative coding is not attempted for this feature. Note also that using the normalized difference method, only the Height feature would have attempted an alternate coding. Features with a high confidence in any of the methods do not attempt alternate codings.

{Temperature:hot , mild (0.4);
Color:blue, green (0.85);
Height:tall, very tall (0.05)}

Example 4.2 An information stream forwarded from the CC to the SL

4.4.5 Weighted Multiple

The weighted multiple method is the only method which does not use the output values of the output nodes in the computation of the confidence. Instead, the number of times an output node has been reinforced is used. To prevent this number from growing too large, the value is input in a decreasing function. Equation 4.9 shows this method.

$$c_j = \ln (r_j), \qquad \text{(eq. 4.9)}$$

where c_j is the confidence of output node j, and r_j is the number of times output node j has been reinforced. The natural logarithm is applied to the value r_j so that a useful coding may still be considered as a coding to split.

This method has the intuitive appeal of giving a high confidence to the codings which have proven useful in the past. However, a problem occurs that once a feature has a coding split, that coding may continue to split itself. Consider the case, that Temp:hot has been reinforced 30 times, Color:blue has been reinforced 25 times, Height:tall has been reinforced 15 times, while Height:very tall has also been reinforced 15 times. Anytime the feature Temperature is coded to "hot", the feature Height always will have a lower confidence. An alternative coding for the feature Temperature is never considered. Thus Temperature looses the opportunity to learn by adjusting coding boundaries. Once the feature Height has a new coding created, say coding "short," the output node which represents "short" has never been reinforced. Thus this coding continually has a low confidence which determines the opportunity to learn. If the coding to split is determined by low confidence, then the feature Height is the only feature which creates a new coding. However, at times, it may be desirable to have only the newly created nodes learn.

4.4.6 Activation Range

This final method for calculating confidences uses the output value of the winning node, the number of codings the feature has, and the range that the feature's values cover (prior to normalizing). The formula is given below in equation 4.10

$$c_j = \mathrm{ACT}(o_j) + \frac{\mathrm{RAN}}{\max_i - \min_i} + \mathrm{NUM}(n_i), \qquad \text{(eq. 4.10)}$$

where c_j is the confidence of output node j, o_j is the output value of node j, $\max_i$ and $\min_i$ are the maximum input value and minimum input value of the i^{th} continuous input feature prior to normalization, n_i is the number of discrete codings for the i^{th} input feature, and ACT, RAN, and NUM are user defined parameters.

This method attempts to combine the notions of how close an input value is to the prototype of the codings, with the relative width of the coding. The first term in the equation simply determines how close the input value is to a coding prototype. The second and third terms determine how many codings are covering the entire range of the feature. Thus if the output value of output node j is high, and the input feature represented by input node i has a small range, and the i^{th} feature has a large number of discrete codings, the confidence, c_j would be high. Conversely, if the output value of the output node is low, and the coding is a broad or wide coding relative to the potential input values, then the feature would have a low confidence.

By using parameters to effectively weight each term in the equation, any term can be removed from the equation. A number of experiments have been attempted with a number of different values for each parameter (including zero). The results bare out that each term is important and requires a parameter value greater than zero. As is discussed in the next chapter with the other simulation results, optimal values for these parameters are ACT = 0.0001 and RAN = NUM = .5. The very small value of the constant ACT is due to the fact that output values for the nodes of a given output cluster are not normalized in the activation range method.

4.4.7 Methods to Determine Which Coding to Split

Four selection methods determine the feature for which a new coding is created. Each method is discussed below. Selection of the feature which has a coding created is critical for proper learning. Each method can be evaluated by at least three factors.

- Which confidence methods can be used with a given split method.
- Does one feature continually have codings split, while others do not.
- Are the codings stable and useful for a number of instances.

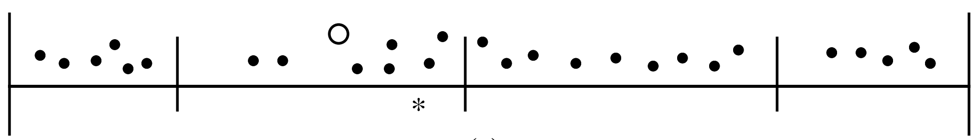
4.4.7.1 Minimum Confidence

The minimum confidence method simply selects the feature with the lowest confidence value. Any of the above methods for calculating confidence can be used in combination with this method and most work equally well. The weighted multiple (section 4.4.5) method however, does not work well in combination with this method. Using the weighted multiple method, the same feature tends to dominate the learning. In some cases, numerous codings would be created for a feature. When too many codings are created, the generalizing ability of the system suffers. Using this method with any confidence method based on output values (including the activation range method) creates a proper balance of stable codings for each feature. Typically two to four codings are created for each feature. This method balances the learning across the features also. The number of codings in each feature is approximately equal. Further, the codings are created rather slowly, giving the CC time to find proper boundaries between codings.

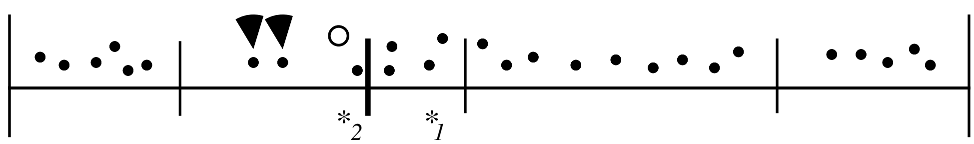
4.4.7.2 Minimum Codings

The minimum codings method splits the coding with the fewest number of codings. Because it is not based on confidence metrics, there is no restriction on the combinations of confidence metrics used. This method creates a balance of codings. However, the coding often is created for the wrong feature. If a given feature has the fewest number of codings, but the input value for that feature is very close to a prototype, that feature is still selected as the feature for coding creation. Thus two codings exist virtually on top of one another. This renders both codings unstable, as inputs previously coded in the original coding are now miscoded in the new coding. Because the prototypes are very near, only confidence methods which find input values near the coding boundaries (normalized ratio difference, and threshold difference from above) are useful to adjust the coding boundaries. To see this, simply observe that a value in between two very near prototypes creates a very high output value for both output nodes representing the prototypes. If the confidence is based on these values instead of the difference between these two values, both codings would have a high confidence, and the feature would never try an alternative coding.

In addition to the problem of learning proper codings for the values in between the codings, this method illustrates a limitation of the coding abilities of the CC. This limitation is shown in figure 4.7. In figure 4.7a the input value for the given feature is shown as the hollow circle, the “*” below the value line represents the prototype of that particular coding. Because this feature has the fewest number of codings, the minimum coding method selects this feature for new coding creation. The prototype for the new coding is set at the value of the input feature, namely the hollow circle. This new coding and prototype is shown figure 4.7b. The boundary between the two codings as defined by the inhibition is shown in bold in figure 4.7b. This boundary may be adjusted to potentially any point between the two prototypes. However, the two values (pointed at by arrows) to the left of the new prototype “$*_2$” can never be coded with prototype “$*_1$” again. The boundary between the two codings is constricted to be between the two prototypes. Put another way, the boundary can never venture to the left of the “$*_2$” prototype to include the marked values. This limitation always exists, but the other coding selection methods tend to give better results in light of it.



(a)



(b)

Figure 4.7 Potential problem of codings with prototypes too near

4.4.7.3 Weighted Minimum Confidence

The weighted minimum confidence method is a compromise between the two selection methods presented thus far. As opposed to selecting the feature with the fewest number of codings, regardless of confidence values, this method allows features with few codings, but high confidence to be passed over in the selection process, by weighting the confidence with the minimum number of codings. This effectively eliminates the problem of having two prototypes too near.

4.4.7.4 Minimum Confidence Threshold

This method is similar to the minimum confidence method discussed above in section 4.4.7.1, the difference being that all features with a confidence below a certain threshold are selected for new coding creation. The value of the threshold determines to a great extent the effectiveness of this method. Using of dynamic threshold gives more consistent results. This is accomplished by selecting any feature with confidence within 10% of the minimum confidence. Using this method, a balance of codings results. By allowing a feature close to the minimum confidence to be selected in addition to just the feature with the minimum confidence, stable codings are created faster. In fact this method typically created close to the same codings as the minimum confidence method with one or two fewer training iterations. Occasionally the CC may create coding too fast, causing a ripple effect resulting in numerous unstable codings. This occurrence is rare, based on empirical testing.

4.5 Analysis of CC

The CC is a parallel algorithm. Using a neural architecture, each node can in parallel calculate its activation and final output. In the execution phase, this results in a constant time bound. Likewise, using a parallel neural architecture in training each node can initialize its appropriate inhibitory weights to the new node. The inhibitory weights from the new node has time complexity $O(n)$, where n is the number of output nodes in the cluster in which the new node is created.

If the algorithm is executed on a sequential machine, the time bound is polynomial. Finding the codings for both learning and execution has time complexity $O(mn+n^2)$ where m is the number is the number of input features, and n is the maximum number of output nodes in any FC. The first term calculates the initial activation of each node, while the second term is the inhibition from all other nodes in the output cluster.

In training, reinforcement of features is bounded by $O(m)$. Adjusting inhibition boundaries takes constant time. Creating a new coding is the most expensive operation and has time complexity of $O(n^2)$. This complexity comes from initializing the inhibitory weights. The worst case time complexity for training is therefore $O(mn+n^2+n^2)$ which reduces to $O(mn+n^2)$.

The codings produced by the CC are dependent on the SL used to generate feedback. A discussion of the codings produced and limitations of these codings is discussed in the next chapter.

Chapter 5

A New Learning Model

This chapter begins with the general framework for a new learning algorithm. After the presentation of the general model, two specific SLs are implemented creating ASOCS-AFE and ID3-AFE. Both learning systems are presented and analyzed. In each case simulation results are discussed.

5.1 SL-AFE Execution

The general model is shown in figure 5.1. In the execution phase, the SL uses the input features to generate an output. The generated output is the output of the system. The algorithm for the execution phase is shown in figure 5.2. Note that this algorithm assumes an instance is available for processing.

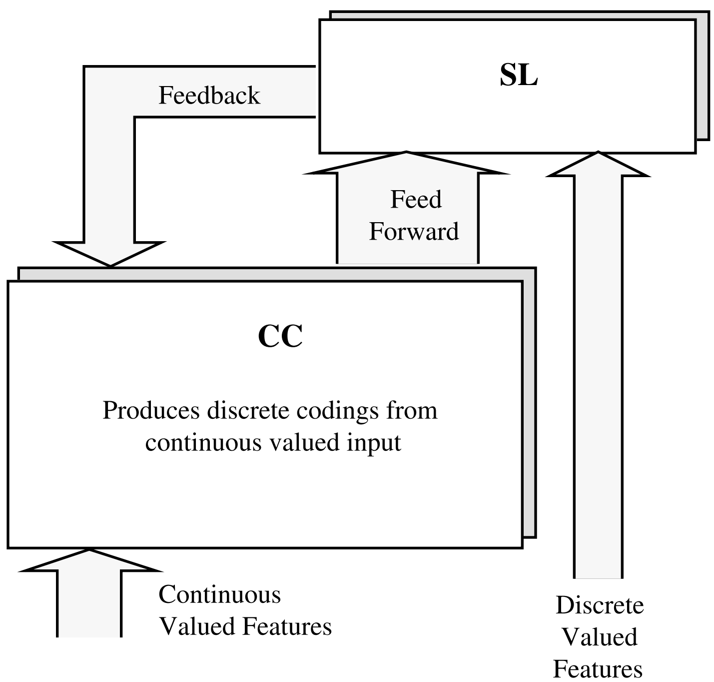


Figure 5.1 A SL-AFE model

SL-AFE execution algorithm
cur_instance:= execution_instance(*input_instance*);
output:= generate_output(*cur_instance*);
end SL-AFE execution algorithm

execution_instance(*instance*)
Copy the discrete features from *instance* to *trial instance*.
Call the CC to code the continuous features and forward codings and confidence information.
Use the first codings as the discrete values for the continuous features and place these values in *instance*.
end execution_instance

Figure 5.2

In the execution phase, the SL-AFE accepts input features. The CC uses the continuous valued features to generate one discrete coding for each feature. The discrete codings are the only information forwarded from the CC to the SL where they are again combined with the discrete valued features. The instance is reconstructed by the SL and now contains only discrete values for all input features. The SL uses the instance to generate an output by the function generate_output which is not defined because it depends upon the details of the particular SL. During execution, no feedback is generated by the SL for the CC.

5.2 SL-AFE Training

The algorithm for the training phase of the SL-AFE is shown in figure 5.3. This figure is general, with many details of the algorithm dependent on the specific SL. The training algorithm can be broken up into three sections. First, initial codings may be found using a coarse-coding technique such as BRACE or a nearest neighbor technique [18][7]. These initial codings may then be used for some learning by the SL without generating feedback. Second, the SL iterates through the training set generating feedback for the CC. Third, the SL may perform some additional learning after the iteration and feedback.

While the SL iterates through the training set, it creates a number of *trial instances*. A trial instance is formed by using different codings for the continuous features. The first trial instance is formed using the first codings for all continuous features. The SL uses the trial instances to produce an output. While the output of the SL is incorrect, the algorithm systematically tries other trial instances which continues until a trial instance yields a correct output, or all options for creating trial instances have been tried.

In SL-AFE of figure 5.3, the variable *trial_instance* stores each of the trial instances created as a vector of instances. The index variable *k* is used to keep track of which trial instance the algorithm is currently considering. Various actions by the SL-AFE depend upon which trial instance is used to produce a certain output. Specific details of the algorithm are discussed in sections 5.2.1 through 5.2.4.

```
SL-AFE  learning  algorithm
  [1]     coarse-coding();            // if necessary for given SL
  [2]     SL_pre_feedback_learning();     // if necessary for given SL
  [3]     for i:= 0 to max_iterations  {
  [4]       for j:= 0 to number_training_instances  {
  [5]         trial_instance[0] = build_trial_instance(
                                    training_instance[j], 0);
  [6]         for k:= 0 to number_continuous_features  {
  [7]           output:= generate_output(trial_instance[k]);
  [8]           if (output = training_instance_output)  {
  [9]             if (k = 0)  {
  [10]              reinforce_codings();
  [11]              SL_learning_correct_response(trial_instance[0]);
                  }  // end if k=0
  [12]            else   {          //  k≠0, output correct
  [13]              adjust_boundaries();
  [14]              SL_learning_correct_response(trial_instance[k]);
                  }  // end else k≠0
  [15]            break;            // goto next training instance
                }  // end if output=training_instance_output
  [16]          else  {       // output≠training_instance_output
  [17]            SL_learning_incorrect_response(trial_instance[k]);
  [18]            trial_instance[k+1]:= build_trial_instance(
                                        trial_instance[k],  k+1);
  [19]            if (trial_instance[k+1] = Null)  {
  [20]              if (trial_instance[0] is learnable by SL based
                        on discrete features)
  [21]                SL_learning(trial_instance[0]);
  [22]              else  {
  [23]                split_codings();
  [24]                new_instance:= build_new_instance();
  [25]                SL_learning_new_instance(new_instance);
                    }  // end else split a coding
  [26]              break;
                  }  // end if trial_instance[k+1]=Null
                }  // end else output[k]≠training_instance_output
              }  // end for loop (k)
            }  // end for loop (j)
          }  //  end for loop (i)
  [27]  SL_post_feedback_learning();              // if necessary for given SL
 end  SL-AFE  learning  algorithm
```

Figure 5.3 The general SL-AFE learning algorithm

The coarse coding in line [1] of the continuous features may be necessary for any learning that the SL performs prior to generating feedback. In section 5.4, ID3-AFE uses coarse coding to give the ID3 algorithm codings to which the gain function is applied. Applying the gain function and constructing an initial ID3 tree is an example of necessary learning prior to generating useful feedback. Through the use of iteration, the SL-AFE can train on each instance in the training set a number of times. The first trial instance is constructed on line [5] by the algorithm build_trial_instance (figure 5.4).

```
build_trial_instance(trial_instance, trial)
  if (trial = 0)      {
      Copy the discrete features from instance to trial instance.
      Call the CC to code the continuous features and forward
      codings and confidence information.
      Use the first codings as the discrete values for the
      continuous features and place these values in
          trial_instance.
      return (trial_instance)
      }  // end if trial = 0
  else  {         // trial ≠ 0
      if (trial > 1)
          change_last_coding(trial_instance);       // sect. 5.2.2
      if (change_next_coding(trial, trial_instance) =
                                          Empty_Feature_Set)
          return (Null);
      else
          return (trial_instance);
      }  // end else trail ≠ 0
 end  build_trial_instance
```

Figure 5.4 Build_trial_instance algorithm

The first trial instance is used by the SL to generate an output. The output is then compared to the training instance output.

5.2.1 First Trial Instance SL Output is Correct

If the first trial instance can be used to generate a correct output from the SL, feedback is generated to reinforce the first codings for each of the continuous features. The test on line [9] of the SL-AFE learning algorithm determines if the first trial instance produced the correct output. Reinforcing the codings which produced the correct output makes those codings more likely to occur when a similar training instance is processed. The algorithm to reinforce codings is shown in figure 5.5. The SL may also learn when its output is correct. This however, is not shown in algorithmic form since it is specific to the SL being used.

reinforce_codings()
  for each continuous feature *i*

$$w_{i,j} = w_{i,j} + \frac{w_{i,j} - m_i}{r_j}$$

 end reinforce_codings

Figure 5.5 Reinforce_codings algorithm

5.2.2 First Trial Instance SL Output is Incorrect

When the output from the SL does not equal the training instance output for the first trial instance, the SL may learn making the first trial instance less likely to occur. In addition, the SL uses the algorithm build_trial_instance of figure 5.4 to construct additional trial instances. The algorithm change_last_coding (figure 5.6) is used to change the coding of a selected feature back to its original coding. The feature selected is the feature which is using a second coding in the current trial instance. The algorithm change_next_coding (figure 5.6) selects the feature from a set (which is constructed if it doesn't already exist) with the minimum confidence. The variable *feature* is a global variable which stores which feature has just been changed. This feature is referenced also in changing coding boundaries.

change_last_coding(*trial_instance*)
Find *feature* in *trial_instance* and change its value to its original first coding.
Delete *feature* from *S*.
end change_last_coding
change_next_coding(*trial*, *trial_instance*)
if (*trial* = 1) {
Construct feature set *S* by including all continuous features with confidence below the threshold (if the confidence method uses a threshold), or by including only the feature with minimum confidence.
} // end if *trial* = 1
if ($S \neq \emptyset$) {
feature:= feature in *S* with minimum confidence.
change coding of *feature* to second coding.
} // end if $S \neq \emptyset$
else // $S = \emptyset$
return (Empty_Feature_Set);
end change_next_coding

Figure 5.6 Algorithms used to build trial instances

If a second trial instance must be constructed, the algorithm change_next_coding is called from build_trial_instance and constructs a set of features *S* such that each element is *S* is a continuous valued feature whose confidence is below the threshold if the confidence method uses a threshold. The set *S* is constructed the first time it is needed (to build the second trial instance). If the confidence method does not use a threshold, then *S* contains the single feature which has the lowest confidence. For a feature to appear in *S*, it must appear in the training instance. Thus a feature with an unknown input value cannot be in *S*. Once *S* is constructed, the feature with minimum confidence is selected. From the first trial instance, this feature changes its coded value to the second possible coding forwarded by the CC creating the second trial instance. Using this second trial instance, the SL finds a new output. This new output is compared to the training instance output. If $S = \emptyset$, there are no features with which to construct trial instances.

5.2.3 Second or Subsequent Trial Instance SL Output is Correct

If the output generated by the second or any subsequent trial instance equals the training instance output, then the CC makes the second coding of the feature which was changed to create the second trial instance more likely to occur. This is accomplished by generating feedback for the CC to adjust the coding boundary between the first and second possible codings of that feature. The else of line [12] in the algorithm SL-AFE of figure 5.3 uses the variable *k* to detect if the trial instance producing the correct output is not the first trial instance. The algorithm to adjust coding boundaries is shown in figure 5.7.

adjust_boundaries()

$$i_{j,k} = i_{j,k} - (1 - \frac{o_j}{o_k})$$

$$i_{k,j} = i_{k,j} + (1 - \frac{o_j}{o_k})$$

end adjust_boundaries

Figure 5.7 Adjust_boundaries algorithm

The SL at this point may also learn to make the trial instance producing the correct output more likely to occur. This again would depend upon the specific SL being used.

If a third (or any subsequent) trial instance is created, the algorithm change_last_coding of figure 5.6 changes the feature which was changed to create the previous trial instance back to its original coding and removes that feature from the set *S*. For any trial instance, only one feature is changed from its original coding. As each trial instance is created, an output is found for that instance.

5.2.4 No Subsequent Trial Instance SL Output is Correct

If no trial instance gives the correct output from the SL, then the SL attempts to learn the first trial instance based on its discrete features. If the discrete features are insufficient for the SL to learn the first trial instance, then feedback may be generated to split one or more codings depending upon the split method being used and characteristics of the SL. If the discrete features of one training instance may match the discrete features in another training instance, and the two training instances imply different outputs, then the SL cannot learn the two instances based on the discrete features. Figure 5.8 shows the algorithm to split a single coding.

split_coding()
select features to split by split method being used.
for each feature selected {

$$i_{k,j} = \overline{o_k} * (\frac{-1}{p})$$

$$i_{j,k} = -\overline{o_k}(1 + i_{k,j})$$

$$w_{i,j} = m_i$$

}

end split_coding

Figure 5.8 The algorithm to split codingss

After the feature has been split, a new instance is constructed using the algorithm in figure 5.9.

build_new_instance()
Construct an instance using the discrete codings from the training instance, using the first codings from all features except those with new codngs. The features with new codings use the new coding values.
Return the new instance.
end build_new_instance

Figure 5.9 The build new instance algorithm

The SL now attempts to learn the new instance constructed by the build_new_instance algorithm.

The first trial instance should be learned based on its discrete features because the proper codings of the discrete features are given as input, and do not need to be discovered. Discovery of codings for the continuous valued features is at best a guess. If the instance can be learned based on its discrete features, and without creating additional codings, it is done. Learning based on the discrete features is illustrated in both ASOCS-AFE and ID3-AFE. The SL may also learn based on the incorrect responses of one or more trial instances.

Creating additional codings is used as a last resort because it can result in some unlearning of previously learned information. Suppose that the input of Temperature:89˚F is coded as Temperature:hot in training instance 1. This coding (along with the rest of the instance) is learned by the SL. A later instance in the training set then creates the coding "mild" for the feature Temperature into which the value 89˚F is coded. Thus the previous learning by the SL no longer applies to training instance 1. A new instance may need to be learned by the SL using the "mild" coding of the Temperature feature next iteration when training instance 1 is presented. The SL may also determine that coding 89˚F into "hot" instead of "mild" is a better coding. This could be determined by trying an alternate coding for the feature Temperature. Iteration makes either type of learning possible, but not guaranteed. The proper number of iterations must be determined by empirical testing. Because the CC never destroys a coding once it is created, new codings are only created when the SL can do nothing else to learn the instance. Future work will include the ability to destroy a coding or merge two codings into one.

5.2.5 An Example

This example, illustrates how the SL-AFE learning algorithm executes. Prior to iterating through the training set, the continuous features may be coarse-coded into a small number of codings. Further, the SL performs any learning based upon the coarse-codings and other information available from the training set. Now, suppose that the following instance from the golf data set of chapter 1 is to be learned by the SL-AFE. The training instance is.

Temperature:99˚F Humidity:65% Windy:False Outlook:Cloudy -> N

The features temperature and humidity are input to the CC, while the inputs windy and outlook are input directly to the SL. Suppose the stream forwarded by the CC is.

{Temperature:hot, mild, 0.83; Humidity:high, low, 0.55}

The SL constructs the first trial instance by line [5] of the SL-AFE training algorithm as

Temperature:hot Humidity:high Windy:False Outlook:Cloudy (Trial 1)

Using this instance, the SL generates an output and compares this output to the output of the training instance. If the SL generated output were N (the output of the training instance), then the variable k is used to determine which trial instance produced the correct output. If the first trial instance is used to produce the correct output ($k = 0$), then feedback to reinforce the codings "hot" and "high" would be sent to the CC, and the SL may learn. If the SL generated output were P (not equal to training instance output), then the SL would construct a second trial instance as in line [18] of the SL-AFE algorithm. The second trial instance is constucted by considering an alternate coding for a feature. The feature selected for an alternate coding is Humidity because its confidence is lower than the confidence of Temperature. The second trial instance is

Temperature:hot Humidity:**low** Windy:False Outlook:Cloudy. (Trial 2)

Using this instance, the SL generates another output and compares it to the output specified by the training instance. If the SL generated output were N, then the variable k is used to determine which trial instance is used to produce the correct response. Since this is the second trial instance and second computed output, $k \neq 0$ so feedback to adjust the boundaries between the codings "high" and "low" of the feature humidity would be sent to the CC. If on the other hand, the SL output were Y, the actions would depend on the confidence method being used. If the confidence method does not use a threshold, the set S is now empty since it contained only the single element Humidity. At this point, the SL attempts to learn the first trial instance based on its discrete features. If the discrete features are not sufficient to learn the instance, then feedback to split the humidity coding is sent to the CC. If the confidence method did use a threshold, and the threshold were greater than 0.83, then another trial instance would be constructed by changing the value of the Humidity feature back to its original coding of “high”, and changing the value of the Temperature feature to “mild”. This creates

Temperature:**mild** Humidity:**high** Windy:False Outlook:Cloudy. (Trial 3)

An output is again generated and compared. If the output is correct, feedback to adjust the boundaries between the codings “mild” and “hot” of the feature Temperature is sent back to the CC. If the input is incorrect, an attempt is made to construct another trial instance. The set S is now empty, and the algorithm proceeds as previously stated for $S = \emptyset$. If a new coding is created, then the SL learns the newly formed instance. For example, if the feedback generated were to split the coding “high” of the humidity feature, then the SL learns the instance

Temperature:hot Humidity:**medium** Windy:False Outlook:Cloudy -> N

where **medium** is the new coding created for the humidity feature.

5.3 ASOCS-AFE

An ASOCS-AFE is created by using an ASOCS as the SL in the SL-AFE algorithms above. For reasons discussed in chapter 2, Priority ASOCS is used with the extensions for multi-state discrete features. By creating ASOCS-AFE, ASOCS is extended to allow continuous valued features.

5.3.1 ASOCS-AFE Execution

The execution phase of the ASOCS-AFE is shown in figure 5.10. The algorithm execution_instance is shown in figure 5.2, and the algorithm generate_PASOCS_output is shown in figure 2.7. The instance constructed is checked against the PASOCS network. The matching instance (if one exists) with the highest priority sets the output for the ASOCS-AFE. If there is no matching instance, the PASOCS output is set to no output.

ASOCS-AFE execution algorithm
cur_instance:= execution_instance(*input_instance*);
output:= generate_PASOCS_output(*cur_instance*);
end ASOCS-AFE execution algorithm

Figure 5.10 The ASOCS-AFE execution algorithm

5.3.2 ASOCS-AFE Training

The training algorithm for the ASOCS-AFE is shown in figure 5.11. It differs from the algorithm of figure 5.3 in the following ways. First, ASOCS-AFE does not initially coarse code the continuous features. If no codings exist for a given continuous feature, then the first coding is created using the input value of the continuous feature for the prototype of the created coding. Since there are no other nodes for the particular feature, there is no inhibition. In this way, ASOCS-AFE can create codings as needed. If the first value of a continuous feature is erroneous, or isolated from other values for that feature, then the first coding created for that feature may not be very general. However, it may not be desirable to have a coding containing either an error or an isolated value to contain a number of other values.

ASOCS-AFE does not require any learning prior to generating feedback. The PASOCS network is initially empty. The first instance is then used to create the first node in the network.

ASOCS-AFE learning algorithm

```
for i:= 0 to max_iterations {
    for j:= 0 to number_training_instances {
        trial_instance[0] = build_trial_instance(j, 0);
        match:= match_discrete_features(trial_instance[0]);
        if (match = True) {
            for k:= 0 to number_continuous_features {
                output[k]:= generate_PASOCS_output(
                                        trial_instance[k]);
                if (output[k] = training_instance_output) {
                    if (k = 0)
                        reinforce_codings();
                    else
                        adjust_boundaries();
                    break;
                    } // end if output[k]=training_instance_output
                else {      // output[k]≠training_instance_output
                    trial_instance[k+1]:= build_trial_instance(j, k+1);
                    if (trial_instance[k+1] = Null) {
                        if (output[0] = No_PASOCS_Output)
                            add_instance(0, trial_instance[0])
                        else {          // output[0] set to wrong value
                            split_codings();
                            new_instance:= build_new_instance();
                            next_priority:= compute_next_priority(
                                                training_instance[j]);
                            add_instance(next_priority,
                                                new_instance);
                            } // end else output[0] set to wrong value
                        break;
                        } // end if trial_instance[k+1]=Null
                    } // end else output[k]≠training_instance_output
                } // end for loop (k)
            } // end if (match)
        else {      // match = False (match of discrete features)
            next_priority:= compute_next_priority(
                                                training_instance[j]);
            add_instance(next_priority, trial_instance[0]);
            } // end else discrete features do NOT match
        } // end for loop (j)
    } // end for loop (i)
```

end learning algorithm

Figure 5.11 The ASOCS-AFE learning algorithm

The learning algorithm iterates through each instance in the training set a number of times. For each training instance, a first trial instance is created as in figures 5.2 and 5.4. The function match_discrete_features broadcasts the first trial instance to the PASOCS network along with the test "discrete match" which matches discrete features and is shown in figure 5.12.

```
match_discrete_features(trial_instance)
  broadcast(trial_instance, discrete_match);
  boolean_value:= gather(True flags);      // discrete_features_matched
  return(boolean_value);
 end  match_discrete_features

discrete_match(trial_instance)
  node.discrete_instance_matched:= True;
  for each discrete feature in trial_instance  {
      if (instance.feature.state ≠ Don't_Care)  {
          if  (PASOCS_instance.feature.state ≠ Don't_Care)  {
              if (trial_instance.feature.state ≠
                                  PASOCS_instance.feature.state)  {
                  node.discrete_instance_matched:= False;
                  }  // end if states ≠
              }  // end if PASOCS = Don't_Care
          }  // end if instance = Don't_Care
      }  // end for each discrete feature in instance
 end  discrete_match

True  flags
  if ((left_child.flag = True) OR (right_child.flag = True) OR
          (node.discrete_instance_matched = True))
      pass_to_parent(True);
  else
      pass_to_parent(False);
 end  True  flags
```

Figure 5.12 ASOCS-AFE learning algorithms

If the PASOCS network can match the discrete features, then there is a chance the continuous features may also produce a match. The output of the first trial instance is computed as in the execution phase using algorithm ASOCS-AFE_execution of figure 5.10. If the output of the first trial instance is equal to the output of the training instance, the codings are reinforced as in figure 5.5. PASOCS does nothing when an output is correct, therefore there is no learning by the PASOCS.

If the output produced by PASOCS does not equal the output of the training instance, the first output is saved for later reference, and a new trial instance is constructed as in figures 5.4 and 5.6. The first trial instance and its output are saved because PASOCS has two types of incorrect answers. The network may set the output value to an incorrect value, or the network may fail to match the instance which does not set the PASOCS network to any value. As each subsequent trial instance is created, it is used to generate an output. If any subsequent trial instance produces a correct output, boundaries are adjusted in the CC by the algorithm of figure 5.7.

If no trial instance produces a correct output, an instance is added to the PASOCS network by the algorithm of figure 5.13. The instance to add is determined by the output generated by the first trial instance. If the PASOCS network did not produce an output, no instance in the network matched the first trial instance produced. Therefore, the first trial instance produced is added to the PASOCS network with a base priority.

```
add_instance(priority, instance)
  Add instance to the PASOCS instance set.
  Allocate a free node in the network.
  node.instance_priority:= priority;
  node.instance_output:= training_instance.output;
  node.instance:= instance;
 end add_instance
```

Figure 5.13 Add_instance algorithm

If the PASOCS network sets the output to a valid output value, but the output is incorrect, then one or more codings are split by the algorithm in figure 5.8. The algorithm of figure 5.9 constructs a new instance. This instance is added to the PASOCS network with the priority one greater than the highest contradicting priority which is found by the algorithm compute_next_priority (figure 5.14).

If the PASOCS network cannot match the discrete features, the first trial instance is added to the PASOCS network with a higher priority than any contradicting instance.

```
compute_next_priority(training_instance)
  broadcast(training_instance.polarity, check contradiction);
  priority:= gather(largest contradicting priority);
  return (priority+1);
 end compute_next_priority

check contradiction        // test at node for broadcast
  if (node.output ≠ training_instance.output)
      node.instance_contradicted:= True;
  else
      node.instance_contradicted:= False;
 end check contradiction

largest contradicting priority                    // test at node for gather
  if (node.instance_contradicted = True)
      pass_to_parent(max(node.priority, left_child.priority,
              right_child.priority));
  else
      pass_to_parent(max(left_child.priority,
                              right_child.priority));
 end largest contradicting priority
```

Figure 5.14 Algorithms for finding proper priority

By adding the first trial instance created to the PASOCS network when discrete features are not matched, ASOCS-AFE is capable of learning exceptions based on values of the discrete features.

5.3.3 An ASOCS-AFE Example

The IS shown already exists in the PASOCS system as a result of previous training. The values in parenthesis indicate the priority of each instance. The IS is in table form instead of the PASOCS tree structure for simplicity.

IS

I. Temperature:hot Humidity:high Windy:false Outlook:sunny -> N (0)
II. Temperature:cold Humidity:low Windy:true Outlook:rain -> N (0)
III. Temperature:mild Humidity:high Windy:true Outlook:sunny -> P (1)
IV. Temperature:cold Humidity:high Windy:false Outlook:cloudy -> P (1)

Let the training instance (denoted as NI) be

Temperature:59˚F Humidity:45% Windy:false Outlook:cloudy -> N (NI)

The algorithm build_trial_instance causes the CC to code the continuous valued features. Suppose the CC forwards the stream shown.

{Temperature:cold, mild, 0.65; Humidity:low, high, 0.52}

The first trial instance constructed is

Temperature:cold Humidity:low Windy:false Outlook:cloudy. (Trial 1)

The algorithm match_discrete_features finds that the discrete features of NI match the discrete features of one instance in IS (instance IV), and algorithm continues by generating a PASOCS output. This trial instance is not matched in the PASOCS and PASOCS does not produce a valid output value, so a second trial instance is constructed.

Temperature:cold Humidity:**high** Windy:false Outlook:cloudy (Trial 2)

This instance matches the instance IV in the IS, however the output is incorrect, thus another trial instance is constructed (assuming the confidence method uses a threshold).

Temperature:mild Humidity:low Windy:false Outlook:cloudy (Trial 3)

The third trial instance does not match any instance in IS. There are no other continuous valued features with which to construct alternate instances. Thus the output of the first trial instance is used to determine the proper course of action. Since the first trial instance failed to match any instance in the IS, the first trial instance is merely added to the PASOCS network with a base priority. This gives the new IS.

<u>IS</u>

I. Temperature:hot Humidity:high Windy:false Outlook:sunny -> N (0)
II. Temperature:cold Humidity:low Windy:true Outlook:rain -> N (0)
III. Temperature:mild Humidity:high Windy:true Outlook:sunny -> P (1)
IV. Temperature:cold Humidity:high Windy:false Outlook:cloudy -> P (1)
V. Temperature:cold Humidity:low Windy:false Outlook:cloudy -> N (0)

Let the next training instance be

Temperature:92˚F Humidity:65% Windy:false Outlook:sunny -> P (NI)

The algorithm build_trial_instance causes the CC to produce codings for the continuous features. Suppose the CC forwards the following stream to the ASOCS.

{Temperature:hot, mild, 0.92; Humidity:high, low, 0.45}

The first trial instance is constructed by the ASOCS based on the information in this stream is

Temperature:hot Humidity:high Windy:false Outlook:sunny. (Trial 1)

The discrete features of this trial instance are found to match at least one instance in the PASOCS network. An output is generated by the PASOCS. (Since this trial instance matches instance I, the output contradicts the training instance.) Additional trial instances are formed considering the features with confidence below some threshold. The humidity feature has the lowest confidence, therefore a second trial instance is constructed as shown below.

Temperature:hot Humidity:**low** Windy:false Outlook:sunny (Trial 2)

This instance does not match any instance in the IS, thus no output is generated. A third trial instance would now be constructed using the “mild” coding of the temperature feature, if the confidence method used a threshold, and the value of the threshold were greater than 0.92 (the confidence of the temperature feature). Examination of the IS shows that no instance would match the third trial instance constructed as discussed above. Therefore, the results of the first trial instance determine the course of action. The first trial instance contradicts the training instance and feedback to create a new coding is sent to the CC. The humidity feature is selected as the only feature for new coding creation (this selection process considers the split method being used and perhaps also a threshold). The coding “high” is the first coding of the feature humidity, so this coding is split. For simplicity, the new coding is referred to as “medium”. The instance shown below is added to the IS with priority one greater than the highest contradicting priority.

Temperature:hot Humidity:**medium** Windy:false Outlook:sunny

This addition creates the new IS below.

IS

I. Temperature:hot Humidity:high Windy:false Outlook:sunny -> N (0)
II. Temperature:cold Humidity:low Windy:true Outlook:rain -> N (0)
III. Temperature:mild Humidity:high Windy:true Outlook:sunny -> P (1)
IV. Temperature:cold Humidity:high Windy:false Outlook:cloudy -> P (1)
V. Temperature:cold Humidity:low Windy:false Outlook:cloudy -> N (0)
VI. Temperature:hot Humidity:medium Windy:false Outlook:sunny -> P(1)

The first instance is now properly discriminated from the final instance by the humidity feature and the priority.

5.3.4 Analysis of ASOCS-AFE

ASOCS-AFE handles noisy data different from traditional PASCOS. To produce an incorrect response, the PASOCS network matched the first trial instance. By splitting a coding for a feature, PASOCS has created a discriminating variable (feature), thus the instance can be added with the base priority. In traditional PASOCS, if a training instance is matched by the PASOCS network, but it produces an incorrect response, the training instance is added to the network with a higher priority. This effectively deletes the old instance from the network. If the training instance is erroneous, the traditional method replaces a valid instance with an erroneous instance. Obviously, this is undesirable. If on the other hand, the old instance were erroneous, adding the new instance with a higher priority causes the PASOCS to forget the error.

The ASOCS-AFE approach is different. By creating a new coding and instance, then adding it to the PASOCS network, ASOCS-AFE never forgets the error. However, if the training instance is erroneous, the ASOCS-AFE approach is desirable since the correct older instance is not destroyed. Erroneous values in continuous features have a coding, but could be unlearned through adjustments to the inhibition. Erroneous values in continuous features may not have the same negative consequences. If a discrete feature has an erroneous value, then ASOCS-AFE may treat it the same as traditional PASOCS, or ASOCS-AFE may create a new coding for a continuous feature. If the error in the discrete feature causes the PASCOS network to match the discrete features of an instance and then generate an incorrect output, a new coding is erroneously created. If the error in the discrete feature causes the PASOCS to not match an instance, then the error is learned.

5.3.5 Complexity of ASOCS-AFE

The time complexity of ASOCS-AFE learning is polynomial. For each NI, the following actions occur.

- Code continuous valued features
- Construct one or more trial instances
- Generate output for each trial instance
- CC learning
- PASOCS learning

From section 4.5 (assuming a parallel architecture) coding the continuous valued features has a constant time bound. Construction of the trial instances has two steps. First,the set S is created by considering each continuous feature. This step has time complexity $O(m)$ where m is the number of continuous input features. The second step is the actual construction of the trial instances. The constuction of each trial instance takes constant time. However, the number of trial instances created is bounded by m. Thus the construction of the trial instances has total time complexity of $O(m + m)$ which reduces to $O(m)$.

From section 2.6, the time complexity to compute a PASOCS output is $O(\log_2 p)$, where p is the number of nodes in the PASOCS network. From section 4.5, the learning time complexity of the CC is $O(n)$, where n is the number of output nodes for a given FC. Finally, from section 2.6 the time complexity of PASOCS learning is bounded at $O(\log_2 p)$. Collecting this gives $O(n + \log_2 p)$

5.3.6 Simulation of ASOCS-AFE

The ASOCS-AFE model has been simulated. The purposes for developing the simulation are two-fold. First, it is important to show that the model has results comparable to other learning models. Second, through analysis of the simulation, characteristics can be learned which lead to improving the model. Simulations have been run to determine two things. First, what combinations of confidence methods and split methods give the best results and worst results. Second, using the best combinations found above, how does ASOCS-AFE do compared to other inductive learning models. The data used for the simulations comes from [13]. Results are shown here for only three data sets due to the number of simulation runs for each data set.

5.3.6.1 Combinations of Methods and Their Performance

Table 5.1 shows results of the iris data set for all possible confidence methods and two of the four possible split methods. The two methods for splitting codings represent the best method and the worst method. The numbers represented give the accuracy as measured by the percentage of the test set correctly classified.

	Minimum Confidence	Minimum Codings
Normalized Ratio	90.1	62.7
Normalized Difference	83.5	64.6
Normalized Ratio Threshold	92.3	58.3
Normalized Difference Threshold	81.7	53.3
Weighted Multiple	61	42.2
Activation Range	94.6	68.3

Table 5.1 Selected simulation results on the iris data set

Results for the weighted minimum confidence split method were computed only for the activation range confidence method, and are not shown in the figure due to the similarity with the minimum coding method. The weighted minimum confidence method is an attempt to overcome the shortcoming of the minimum coding method (potentially placing a prototype very near an existing prototype), but does not greatly improve the results of the minimum coding method. This is due to the similarity to the minimum coding method. Finding the best feature to create a new coding has little to do with the number of codings of each feature.

Results for the minimum confidence threshold method are also not shown because those results are consistently 3 to 10 percent lower than the results for minimum confidence using only a guess as the proper value for the threshold. If the threshold is too high, too many new codings are created, and the results drop dramatically.

The minimum codings split method performs the very worst of any split method, while the minimum confidence split method gives the best results. In general, for data sets used in the simulations, the continuous values which are near prototypes should be more confident, while the continuous values away from coding prototypes are better candidates for new coding creation.

The confidence methods which are based on the final output value of the output nodes give better results. This is due to the principle "that like things should be coded in like codings". In other words, values near each other should be coded in the same coding.

5.3.6.2 Simulation Results

The results of table 5.2 are obtained using the activation range or normalized ratio threshold as the confidence method, and minimum confidence as the split method. After determining that these combinations consistently performed better than the others, the simulation was applied to additional data sets using these combinations. These results are the best of the two approaches, although the accuracy is consistently close. The information is averaged over a number of runs with different instances in the test set.

Data Set	No. Instances	No. Train Set	No. Test Set	Train Incorrect	Test Incorrect	Test Correct	Accuracy (%)
Glass	214	176	38	5.5	8.4	18	47
Iris	150	120	30	0	0	28	94
Hepatitis	155	120	35	2.2	5.8	24	67

Table 5.2 Results of the ASOCS-AFE simulation

The Data Set column gives the name of the data set. The No. Instances column gives the number of instances in the data set. The No. Training Set and No. Test Sets columns state how the data set is partitioned. For example, for the iris data set, there are 120 of the 150 instances in the training set, leaving thirty instances in the test set. Each run uses a different test set of equal size. The Train Incorrect column states the number of instances in the training set which are misclassified in the final iteration. While the Test Incorrect column gives the number of instances of the test set misclassified. Recall that PASOCS may not set the output. Misclassification means that PASOCS set its output to the wrong answer. The values in the Train Incorrect and Test Incorrect column reflect the number of instances for which PASOCS **set** the output to an incorrect value. The Test Correct column shows the number correctly classified of the test set, and the Accuracy column gives the same result as the percentage of the test set correctly classified.

The results shown in figure 5.20 are slightly lower than other learning models [1]. The results for the iris data set are within 1% of other popular learning models. The results for the hepatitis data set are about 10% lower than the best of other learning models, but better than some of the worst. The results for the glass data set are 20% lower than the best. The section below discusses the results and suggests improvements.

5.3.6.3 Discussion of the Results

The iris data set is considerably easier to learn than the other two data sets. This data set has 50 instances in each of three possible output classes. All features are continuously valued. The ASOCS-AFE does not misclassify any instances in this data set. The errors result from instances in the test set which after coding from the CC, do not match any instance in the PASOCS. Recall that PASOCS does not generalize. Results in all three data sets could be improved by using a PASOCS model which incorporates generalization capabilities.

The hepatitis data set has 123 instances in one of the two output classes, with 32 instances in the other. The test sets created did not seek to maintain the proportion of the data set. The test set typically had 12 instances from the "die" class (originally 32 instances) and 20 instances from the "live" class. The misclassifications almost without exception were instances of the "die" class being misclassified in the "live" class. The majority of training classes are in fact "live", thus ASOCS-AFE has minor difficulties learning the exceptions to the majority cases. Again however, most of the errors with this data set are the result of PASOCS not generalizing when it fails to exactly match an instance.

The glass data set has seven possible output classes. The first 170 instances in the data set are examples of the first three possible output classes (approximately 55 to 60 of each class). The final 4 classes have little representation in the data set. The training set created had examples with all seven classes, as did the test set. This created similar problems and results as in hepatitis data set. However, the hepatitis data set has only 6 continuous valued features of 19 total features. The glass data set has 10 features, all of which are continuously valued. Obviously the codings produced by the CC are not as reliable as the known codings of the discrete variables in the problem. Therefore, the glass data set is very difficult for ASOCS-AFE

5.3.6.4 The Number of Codings Created and Instances

The number of codings created for a continuous valued feature and the number of instances in the PASOCS are related. If a large number of codings is created, then the number of instances grows. In the most successful runs of all the data sets, the number of codings and instances remained low. For the iris data set, typically two or three codings are created and the PASOCS has an average of thirteen instances. In the hepatitis data set, as many as four codings may be created for each of the continuous features. In these cases, the numbers of instances would grow. The best runs with the hepatitis database did not create more than three codings for any continuous feature and formed approximately 35 instances. In the worst cases, the hepatitis data set created four codings for some of the continuous features and as many as 60 instances.

The order of the training set effects the order of coding creation. However, the results appear to be very similar. This is primarily due to iteration. By using iteration, the oddities of order in the data set are removed.

5.3.6.5 The Shortcomings of ASOCS-AFE

The ASOCS-AFE model has a number of shortcomings. One shortcoming has been discussed as the inability of PASOCS to generalize. Even more critical is the restrictive nature of confidence metrics and how the current confidence metrics are computed. Everything the CC does is based on feedback from the PASOCS. What the PASOCS does is however, based on confidence. Better confidence metrics will generate better feedback which in turn will generate better codings in the CC.

Related to the need for improved confidence methods, is the idea of improving the responsibility for incorrect outputs. A new coding is created if no trial instance matches the training instance. The feature selected for new coding creation is typically based on confidence. Improved methods are needed for determining which feature should be split.

Finally, the coding of one continuous valued feature could have some bearing on the coding of another continuous valued feature. For example, the coding of 98˚F may be effected by the 99% humidity. This may be coded as "very hot and humid". Currently the CC considers all continuous features to be independent, and allows the PASOCS to form combinations. Techniques to allow the coding of continuous valued features to effect other codings need to be explored.

5.4 ID3-AFE

The ID3-AFE model is created by connecting the CC of chapter 4 with the ID3 supervised learning model of chapter 3. By creating the ID3-AFE, ID3 can now be applied to continuous valued features.

5.4.1 The ID3-AFE Execution

The ID3-AFE execution algorithm is shown in figure 5.15. The function generate_ID3_output processes the tree as described in section 3.1. Now however, continuous valued features may be the test at any node of the tree.

ID3-AFE execution algorithm
cur_instance:= execution_instance(*input_instance*);
output:= generate_ID3_output(*cur_instance*);
end ID3-AFE execution algorithm

Figure 5.15 The ID3-AFE execution algorithm

5.4.2 The ID3-AFE Training

The training algorithm for ID3-AFE model is shown in figure 5.16.

ID3-AFE learning algorithm

```
coarse-coding();          // if necessary for given SL
SL_pre_feedback_learning();     // if necessary for given SL
for i:= 0 to max_iterations  {
     for j:= 0 to number_training_instances  {
    trial_instance[0] = build_trial_instance(
                              training_instance[j], 0);
    for k:= 0 to number_continuous_features  {
        output:= generate_ID3_output(trial_instance[k]);
        if (output = training_instance_output)  {
            if (k = 0)  {
                reinforce_codings();
                }  // end if k=0
            else   {        //  k≠0, output correct
                adjust_boundaries();
                }  // end else k≠0
            break;         // goto next training instance
            }  // end if output=training_instance_output
        else  {        // output≠training_instance_output
            if (k = 0)  {      // Only do this one time
                if (discrete_feature_not _in_tree)
                    ID3_learning(trial_instance[0]);
                    break;           // to next training instance
                    }
            trial_instance[k+1]:= build_trial_instance(
                                   trial_instance[k],  k+1);
            if (trial_instance[k+1] = Null)  {
                split_codings();
                new_instance:= build_new_instance();
                ID3_learning(new_instance);
                break;
                }  // end if trial_instance[k+1]=Null
            }  // end else output≠training_instance_output
        }  // end for loop (k)
    }  // end for loop (j)
}  //  end for loop (i)
```

end SL-AFE learning algorithm

Figure 5.16 The ID3-AFE training algorithm

Coarse coding is used to give the ID3 system discrete coding values for continuous valued features. Without coarse coding, continuous valued features could not appear in the first decision tree constructed. The course coding method can be of any variety. The methods used for ID3-AFE are discussed in section 5.4.4.

The algorithm then builds the ID3 tree as in the original ID3 algorithm presented in chapter 3. Each instance of the training set is applied to the tree just constructed to determine its output. A first trial instance is constructed by the algorithm build_trial_instance and an output is computed for this instance. If the output of the tree matches the output of the training instance, then the variable k is used to determine which trial instance produced the correct response. If $k = 0$, then feedback is sent to the CC to reinforce the current codings. If the output of the tree does not match the training instance, then first determine if a discrete attribute exists in the trial instance which is not part of the ID3 tree. The algorithm discrete_feature_not_in_tree (figure 5.17) determines if discrete feature exists which is not part of ID3 tree. If so, add this training instance to the training window through the function ID3_learning (figure 5.18).

discrete_feature_not_in_tree
  if (*trial_instance* in training window)
    return (False);
  If *trial_instance* contains a discrete feature which is not part of the
    current tree.
    return (True);
  return (False);
end discrete_feature_not_in_tree

Figure 5.17 Algorithm discrete_feature_not_in_tree

ID3_learning(*instance*)
  Add *instance* to the ID3 training window.
  If a new coding has been created, set flag to iterate through learning
    algorithm at least one more time. This will give the new coding
    the chance to part of the classifying tree if necessary.
end ID3_learning

Figure 5.18 Algorithm ID3_learning

If a discrete attribute does not exist, or if the training instance is already part of the training window, construct subsequent trial instances as in section 5.2. If any subsequent trial instance gives the correct output, then send feedback to adjust the coding boundaries between the two codings of the feature which used an alternate coding to generate the correct response from the ID3 tree. If all trial instances give incorrect responses, then send a message to the CC to split one or more features as demonstrated in section 5.2 and 5.3. Set a flag to construct a new ID3 at the end of the current iteration. This gives the newly created coding an oppurtunity to be part of the next classifying tree.

Iterate through the training set a number of times. After any iteration, the current tree can be destroyed, and a new tree constructed from the current codings. When the current tree is destroyed, the current state of the CC is saved. Therefore, the potentially improved codings from previous iterations are also saved. Prior to constructing the new tree, each instance in the training window must have its continuous valued features coded by the CC into discrete features used by the ID3 algorithm. Because of learning in the CC on previous iterations, continuous values may be coded in different codings than they were when the previous tree was constructed. Thus, the information gain of a continuous valued feature may improve causing that feature to have more discriminating power. If this is the case, the new tree will give more correct outputs than previous trees.

5.4.3 ID3-AFE Complexity

The time complexity for ID3-AFE exection is polynimial and consists of the time required to create the codings by the CC and the time to traverse the ID3 tree. From section 4.5, the time required to code continuous features is constant (assuming a parallel architecture). The time complexity to find an ID3 output is $O(m)$, where m is the number of input features. This of course gives a time bound for the execution of $O(m)$ for each instance classified.

The time complexity for the ID3-AFE learning phase is found by considering the steps involved in learning. Because ID3 builds one tree each iteration, the analysis can no longer consider only a single instance. For each iteration through the training set the ID3-AFE performs the following functions.

- Build ID3 tree
- For each of n training instances
  - Construct at most m trial instances
  - Compute at most m outputs (one for each trial instance)
  - CC learning
  - Add the instance to the ID3 training window

Section 3.3 shows ID3 requires time $O(m^2n)$ where n is the number of instances in the training set. Each instance in the training set has the following requirements. First, constructing m trial instances is shown (in section 5.3.5) to have complexity $O(m + m)$. Computing the output has time complexity $O(m)$. CC learning is shown in section 4.5 to have worst cast time complexity $O(q)$, where q is the number of output nodes for a given FC. Adding the instance to the ID3 window is constant. Collecting gives $O(m^2n + n(m + m + m + q)$ which is reduced to $O(m^2n + nmq)$.

5.4.4 ID3-AFE Simulation

The ID3-AFE simulation has the same purposes as the ASOCS-AFE simulation. The first purpose is to show the model has acceptable results. The second purpose is to analyze the results of the model for potential shortcomings and improvements.

The same data sets from the UC Irvine repository of machine learning databases are applied to the ID3-AFE simulation. The results are somewhat better using the ID3-AFE model due to the fact the ID3 always gives an output. Table 5.3 shows results for the ID3-AFE simulation.

Data Set	No. Instances	No. Train Set	No. Test Set	Train Incorrect	Test Incorrect	Test Correct	Accuracy (%)
Glass	214	176	38	13	19	19	50
Iris	150	120	30	0.8	1.5	29	95
Hepatitis	155	120	35	4.2	9.7	25	72

Table 5.3 Results of the ID3-AFE simulation

The columns in table 5.3 are the same as in table 5.2. The results are comparable to the ASOCS-AFE simulation. The ability of the ID3 to generalize helps the ID3-AFE to generally better results. The glass data set does not show a great improvement due to the fact that there are not any discrete features. The ID3 algorithm makes good use of the discrete features which exist in problems. In the simulations on the hepatitis data set (which has 13 discrete features, and 6 continuous), the discrete features with known codings were always near the root of the tree. The continuous features were always near the leaves of the tree. Thus the discriminating power of the continuous features was not deemed as good as the discriminating power of the discrete features. This is why the hepatitis data set numbers improved more than the glass data set numbers. The decision trees produced for the glass data set were very deep and bushy. Thus a number of codings were produced for some of the continuous features.

A number of shortcomings of the AFE model are unique to the ID3-AFE model. First among them, ID3 uses an information based criterion (termed gain) to determine which features are good at classifying. This is exactly what the CC confidence metric does. However, the two metrics are not related whatsoever. One problem is that the gain function is a *batch* result, computed for the entire training set a priori. The confidence metric is computed for each input. Therefore a feature with little discriminating power (low gain) over the entire data set may have a very high confidence for a given input. The CC should take advantage of the information used by the ID3 algorithm in the computation of the confidence.

Along this line, a number of variations to the normal functionality of the CC were attempted in the ID3-AFE simulation. In one such variation, the feature selected to be split was not based on confidence, but was found from the ID3 tree itself. The continuous feature nearest the leaf level of the path traversed by the instance was selected as the feature to create a new coding. This however, did not work well since many times a feature not existing in the tree did not learn by creating a new coding. The new trees created after such a split rarely placed the feature with the newly created coding any closer to the root of the tree.

Another variation combined the gain with the confidence. Again the results were not impressive. This is due the aforementioned difference in the gain and confidence.

In all cases, the final accuracy is an improvement over the initial codings created by coarse-coding. Two methods of coarse coding were used. In the first, one half of the data inputs were placed in each of two codings. In the second, the two normalized prototypes were set to 0.333 and 0.667 respectively. The best of the two methods for coarse coding turned out to be the former method. Using this method, the ID3 tree had an accuracy of around 58% on the test set prior to feedback and learning. Using feedback and learning the results improved to over 70% (occasionally reaching 74%). Suppressing new coding creation and using only reinforcement and inhibition the results occasionally reached 70%.

Because of ID3's generalization capabilities, rarely did any input feature create more than one additional coding. Therefore the number of codings remained somewhat constant. Since the number of codings remained constant, the tree size remained constant. The number of new codings created caused the tree to be a little bushier. For the hepatitis data set, the tree typically has 13-17 levels and 100-120 nodes.

The results of the preceding two paragraphs indicate that better metrics need to be developed to determine when to split a coding, and which coding should be split. Methods also need to be looked at to find methods to split one feature due to a combination of instances. In other words, to react to statistics gathered from a batch of instances as opposed to reacting to just one instance.

In a number of simulations, the coarse coding was done by setting the initial prototypes to random values. This in turn gave random results. In one such run, the accuracy quickly improved to just over 80% (for the hepatitis data set). Examination of the initial and final prototypes for the continuous valued features showed that some input features had initial prototypes near the boundary of the possible inputs. Because the inputs are normalized and prototypes are [0, 1], the initial values were near 0 or 1. All continuous valued features had initial prototypes far apart. Only 2 of the 6 continuous valued features created a new coding. The other 4 continuous valued features did not create a new coding. This allowed inhibition to do its very best work. Boundaries between codings (which are restricted to be between the prototypes for the codings) could range almost anywhere. The two codings which were created, were useful codings and added discriminating power to the tree. A related and interesting study would be to set the initial prototypes to zero and one respectively and find the resulting boundaries. This however, has not yet been attempted.

Chapter 6

Chapter 6

Conclusion and Future Work

This thesis has shown how a novel CC can be combined with a SL to dynamically discover critical multi-state features. The CC is capable of learning codings through feedback generated by the SL. The two learning systems fit together to guide the codings of the CC into codings which potentially assist the SL to solve a problem. The model is demonstrated by using two distinct SLs, creating ASOCS-AFE and ID3-AFE. Both of these models have simulations and results are given.

Future research will improve the capabilities of the SL-AFE model. There are at least two areas which could be improved. First, improve the codings produced by the CC. Second, improve the interaction between the two components of the model.

The coding ability of the CC could be improved in a number of areas. The CC needs the ability to destroy useless codings. One potential method to detect when a coding is useless would be to count how many times a coding is reinforced. If the coding is rarely used to construct useful instances, the coding could probably be destroyed. Another possibility would consider the inhibition between two codings. If $i_{1,2} >> i_{2,1}$, then coding *2* has little chance of winning the competition in the CC and being selected as the first coding. It also indicates that using coding *2* as a second coding has not been fruitful in the past. Therefore, coding *2* may be merged with coding *1*.

Better confidence methods to determine how to construct trial instances could also improve the capabilities of the CC. Potential methods might combine current methods with other information such as the inhibition between the two nodes. The SL could also be augmented to feedback its output to the CC. Each output node could then track its first order output class correlation. This could then be used to compute future confidence metrics depending upon the output of the CC. For example, if it is found the coding *1* of a certain input feature is highly correlated with a certain output class of the SL, then a coding into that coding would have a high confidence. If the output of the training instance is highly correlated with a certain coding of a certain feature, this coding should have a very high confidence. This could be extended to perhaps use higher order correlations with the obvious consequences of exponential growth in time and space.

Better methods to determine which feature should have a new coding created are needed. Confidence methods to construct trial instances should be different from methods to determine which feature should be split. Methods to determine which coding to split might use a new metric such as a *responsibility* metric. A responsibility metric would attempt to find the feature most responsible for the miscoding. For example, the CC could be augmented to sense novel inputs or novel combinations of inputs. These inputs could then be considered as potential features for new coding creation.

Currently the SL-AFE constructs new codings based on a single instance. Future methods should contain the ability to create one or two codings based on an entire iteration of the instance set. This may result in creating more useful new coding. In order to do this, the learning algorithms of the CC and SL will require modification. For example, the prototype of the new node could no longer be set to the value of the current input. Another difficulty would include determining the number of features which should have new codings created. Perhaps the feedback for each iteration could be used to compute the usefulness of each coding, feature, and the CC.

The interaction between the two components could be improved by determining when one component is dominating the learning, or when one component is falling behind. For example, if the CC creates new codings too quickly, the SL may not be able to keep up. In such a case, coding creation could be suspended until the SL had absorbed the information.

This metric could also be used to determine which component is best for learning a specfic instance, or group of instances. For example, this metric could decide that the SL could learn a given trial instance based on continuous features which have been coded along with the discrete features. Also, this metric could cause the CC to stop learning, and allow the SL to learn exclusively for a group of instances.

Bibliography

[1] Barker, J. C., Eclectic Learning, PhD Dissertation, Brigham Young University, Provo, Utah, 1994.

[2] Bertelsen, R., and Martinez T. R., Extending ID3 Through Discretization of Contnuous Inputs, Proceedings of FLAIRS-94, 1994, 122-125.

[3] Carpenter, G.A., and Grossberg, S., A massively parallel architecture for a self-organizing neural pattern recognition machine. *Computer Vision, Graphics, and Image Processing*, 1987, **37**, 54-115.

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

[5] Grossberg, S., A neural model of attention, reinforcement, and discrimination learning. *International Review of Neurobiology*, 1975, **18**, 263-327.

[6] Grossberg, S., Competitive learning: from interaction activation to adaptive resonance. *Cognitive Science*, 1987, **11**, 23-63.

[7] Jarvis, R. A and Patrick, Edward A., "Clustering using a similarity measure based on shared nearest neighbors", Nearest Neighbor Norms: NN Pattern Classification Techniques, Ed. Belur Dasarathy, IEEE Computer Society Press, Los Alamitos, California, 1991, pp. 388-97.

[8] Martinez, T. R., Adaptive Self-Organizing Logic Networks, PhD Dissertation, University of California Los Angeles, Los Angeles, California, 1986.

[9] Martinez, T. R. and Vidal, J. J., "Adaptive Parallel Logic Networks", *Journal of Parallel and Distributed Computing,* v. 5, 1988, pp 26-58.

[10] Martinez, T. R., and Campbell, D. M., "A Self-Adjusting Dynamic Logic Module" *Journal of Parallel and Distributed Computing*, v. 11, no. 4, 1991, pp. 303-13.

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

[12] Martinez, T. R., Hughes, B., and Campbell, D. M., Priority ASOCS, To appear in *Journal of Artificial Neural Networks*, 1994.

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

[14] Quinlan J. R., "Induction of Decision Trees", *Machine Learning*, 1:81-106, Kluwer Academic Publishers. Boston, 1985

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

[16] Rumelhart, D.E., and Zipser D., Competitive learning. *Cognitive Science*, 1985, **9**, 75-112.

[17] 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.

[18] 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.