93-33

# AQ17
# A MULTISTRATEGY LEARNING SYSTEM
## The Method and User's Guide

E. Bloedorn, J. Wnek
R.S. Michalski and K. Kaufman

MLI 93-12

February 16, 1994

# AQ17: A MULTISTRATEGY LEARNING SYSTEM: The Method and User's Guide

Eric Bloedorn
Janusz Wnek
Ryszard S. Michalski
Ken Kaufman

Center for Artificial Intelligence
George Mason University
Fairfax, VA 22030

September 1993

# AQ17: A Multistrategy Learning System: The Method and User's Guide


## Abstract

AQ17 is a system for acquiring decision rules or trees from examples and counterexamples and/or from previously learned decision rules. When learning rules, AQ17 uses 1) background knowledge in the form of rules (input hypotheses), 2) the definition of descriptors and their types and 3) a preference criterion that evaluates competing candidate hypotheses. Each training example characterizes an object, and its class-label specifies the correct decision associated with that object. The generated decision rules are expressed as symbolic descriptions involving relations between objects' attribute values. Rule generation is guided by a user-defined rule-preference criterion. The user-defined criterion ranks the importance and tolerance of a number of measures of rule quality including rule complexity, cost and coverage. AQ17 has a number of new features including four for various types of representation space modification. These types include data-driven constructive induction, hypothesis-driven constructive induction, concept-based example removal, and attribute-value discretization. AQ17 also includes a method for generating decision trees from decision rules, as well as a decision rule testing utility for evaluating the predictive accuracy of the generated rules.



**Key words**: Concept learning, Inductive inference, Learning from examples, Constructive induction

## Acknowledgements

The authors thank Jim Ribeiro and Witold Szczepanik for their help with the ANSI C implementation. Witold also contributed to Appendix A describing the new data structures for AQ17.

This research was conducted in the Center for Artificial Intelligence at George Mason University. The Center's research is supported in part by the National Science Foundation under the grants No. IRI-9020266 and CDA-9309725, in part by the Advanced Research Projects Agency under the grant No. N00014-91-J-1854, administered by the Office of Naval Research, and under the grant No. F49620-92-J-0549, administered by the Air Force Office of Scientific Research, and in part by the Office of Naval Research under the grant No. N00014-91-J-1351

## Table of Contents

## 1. Introduction

The AQ17 program learns decision rules or trees by performing inductive inference over a set of teacher-classified training examples and/or a set of initial rules. Training examples are expressed as conjunctions of attribute values. Initial or induced decision rules are logical expressions in disjunctive normal form. The program performs a heuristic search through a space of logical expressions, until it finds a decision rule which best satisfies the preference criterion and that covers all positive examples, but no negative examples. The program implements the STAR method of inductive learning (Michalski, Larson 1983). It is based on the AQ algorithm for solving the general covering problem (Michalski, 1969).

The AQ17 program is an immediate descendent of AQ15. The program AQ15 was written in Pascal by Jiarong Hong and Igor Mozetic as an improvement of the GEM (Generalization of Examples by Machine) program written by Bob Stepp and Mike Stauffer. GEM was written from scratch and was not a modified version of the AQ7-AQ11 series of programs on the Cyber 175 (Michalski 1969; Michalski and Larson 1975, 1978, 1983).

AQ17 includes a number of new features that extend the functionality of previous versions. One change from previous versions is the language in which it is written. AQ17 has been ported to ANSI-C. The most important advantage of this change is that it reduces the need to use limiting system-defined data structures. In particular, the Pascal set structure which under Sun Pascal was limited to a cardinality of 58, was very restrictive as it limited the cardinality of attribute values sets to 58. The ANSI-C version has no preset limitations on the number of variables, the number of values or the number of classes. In addition, the ANSI-C version is on average six times faster than the previous implementation.

The implementation of such features was crucial for the development of the constructive induction capabilities in AQ17 system. Constructive induction systems perform changes in the representation space, i.e. they may change the number of attributes and/or attribute values. The dynamic evaluation of the needs for computing resources and dynamic allocation provides such flexibility.

AQ17 includes the functionality of AQ17-DCI (Bloedorn and Michalski, 1992) and AQ17-HCI (Wnek and Michalski 1992). This provides the user with the ability to do both data-driven (DCI) and hypothesis-driven (HCI) construction of new attributes and hypothesis-driven attribute removal. AQ17 can now also take continuous-valued attributes as input. These input values are discretized using a method based on a $\chi^2$ analysis (Kerber, 1992). Another feature related to the input data involves the format of the example data. In AQ17 all examples are in one data table with the decision variable. There is no longer any special need to build separate tables of examples based on the value of the decision variable. However, AQ17 also accepts data in the old format, in additiojn to the new format. A complete discussion of data formats is provided in section 4.9. Other new functions of AQ17 include a method for dealing with noisy data using concept-based filtration of training data (AQ-NT; Pachowicz, Bala 1992), and a method for learning decision trees from decision rules (AQDT-1; Imam and Michalski, 1993).

One new constraint in AQ17 is the requirement that the declarations of types and variables precede input data (in the form of examples or rules). This requirement is similar to those found in some programming languages where variables must be declared before they are used. The general structure of the input file is as follows:

- Parameters controlling the learning process (parameters, criteria-table)
- Definitions of knowledge types (domain_types, names tables)
- Attribute declarations (variables table)
- Background knowledge in the form of rules (a-rules, l-rules, inhypo tables)

• Data (events tables)

ANSI-C language versions of AQ17 were implemented on three platforms: Sun Workstation, PC-compatible, and Macintosh.

## 2. Knowledge Representation

AQ17 uses the $VL_1$ (Variable-valued Logic system 1) and APC (Annotated Predicate Calculus) (Michalski, 1975, 1983) representational formalisms.

Training examples are given to AQ17 in the form of *events*. Events belong to two or more decision *classes*. *Positive examples* are examples of the class for which rules are being currently constructed. During this construction examples from all other classes are considered *negative examples*. When rules for another class are being generated, the positive and negative example labels are changed accordingly. For each class a decision rule or cover is produced that is complete and consistent. A *complete* rule covers all of the positive examples. A *consistent* rule does not cover any negative examples. The user may provide initial decision rules to the program. These rules are treated as initial hypotheses. Intermediate results during the search for a cover are called hypotheses or partial covers.

Each decision rule is described by selectors. A*selector* is a relational statement and is defined as:

**[TERM RELATION REFERENCE]**

where:
**TERM** is an attribute
**RELATION** is one of the following symbols: <, <=, =, <>, >=, >
**REFERENCE** is a value, a range of values, or an internal disjunction of values.

Selectors state that the attributes in TERM take values defined in REFERENCE. Examples of selectors are shown below:

[color = red, white, blue]
[width = 5]
[temperature = 20...25, 50..60]

A *complex* (or a rule) is a conjunction of selectors. The following are examples of complexes:

[color = red, white, blue] [stripes = 13][stars = 1..50]
[width = 12] [ color = red,blue]

A *cover* (or a hypothesis) is a disjunction of complexes. The following is an example of a two complex cover.

[color = red, white, blue] [stripes = 13][stars = 50] v
[color = red, white, blue] [stripes = 3][stars = 1]

A cover is satisfied if any of its complexes are satisfied, while a complex is satisfied if all selectors in it are satisfied. A selector is satisfied if all attributes and expressions in it actually take one of the specified values. The example cover just given can be interpreted as follows: an object is a flag if:

1) Its color ir red, white, or blue and it has 13 stripes and 50 stars on it, or
2) Its color is red, white, or blue and it has 3 stripes and 1 star on it.

## 3. Algorithm

### *3.1 Control Flow*

Figure 1 is a flowchart of the AQ17 program. It shows the integration of seven modules under the control of the user. The core of the AQ17 program is the AQ Learning module. This module is extensively described in the following sections. For additional information see (Hong, Mozetic and Michalski, 1986). Modules DCI, HCI, SCALE, and NT modify the representation space. The DT Module provides a capability for generating a decision tree from already learned rules. A brief description of each module is included in this report.

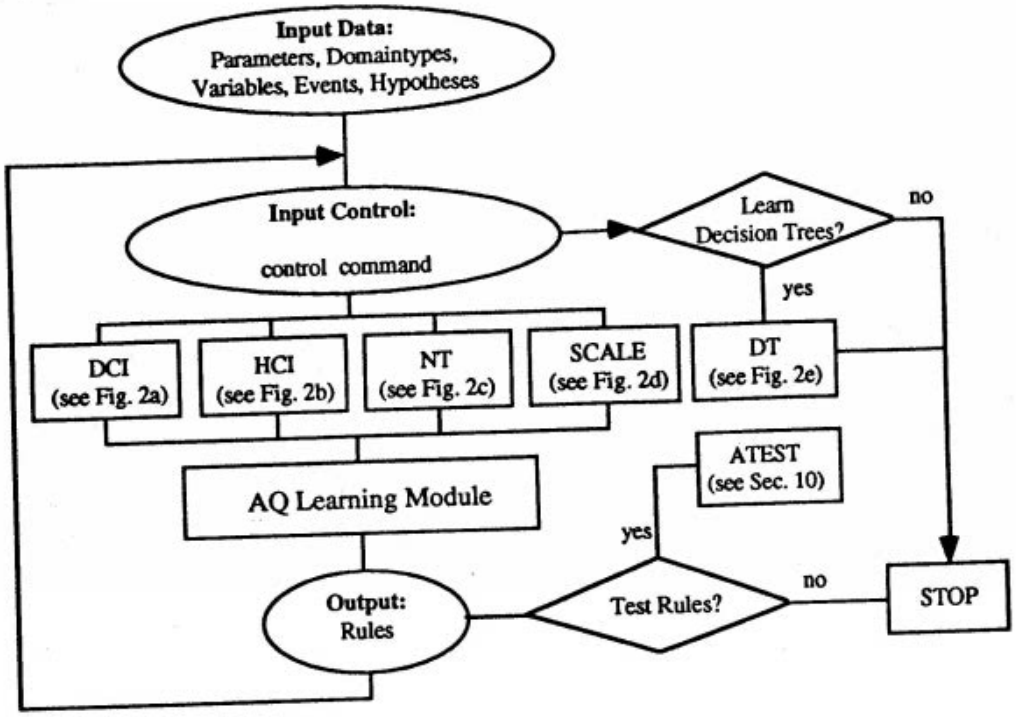



*Figure 1.* AQ17 Algorithm

### *3.2 General Input Format*

Input to AQ17 and all tools in the AQ family is in the form of a set of relational tables (Hong, Mozetic, Michalski, 1986). Relational tables have three parts 1) a table-name, a header, and a list of tuples. Legal table-names, headers and tuples are described below. All tables must be separated by at least one blank line. Values within a header may be separated by spaces or tabs. In many cases some columns are optional (e.g. cost column in variables table). If an optional column is not used then the default value is assigned. Table columns may also be ordered differently than is presented here in some cases. For example in the domaintypes table levels may come before (to the left) or after (to the right) the cost column.

Data tables in AQ17 can be provided in two formats. In the old format, data was formatted using *events* tables (section 4.9). In the new format data can be provided to the system in a simpler comma-delimited ASCII table. In this table, the value of the decision variable (class) is simply another column in the table. Both data formats are described in section 4.9. If the old format is used, data should be stored in a filestem.train file. If the new format is used the table should be stored in a filestem.data file.

In AQ17, the input tables are split into a number of different files: parameters, names, variables, data and testing. Each input file may consist of one or more tables. For example, the names file may consist of a variables table and multiple -names tables (see descriptions below), while a parameter file will consist of just a *parameters* table. Files are identified by a file suffix. The format for the name of a data file (new format) is *filestem*.dat. The filestem may be any valid unix filename. A complete list of files, their suffix, and content is given below:

| Name | Suffix | Content |
|---|---|---|
| domain | .domain | variables definitions: -names, variables, domaintypes tables |
| training data (new) | .data | single table of comma delimited data. Class variable in last column. |
| training data (old) variable: | .train | multiple tables of data divided according to value of class -events tables. |
| test | .test | labelled testing examples: -tevents tables. |
| learning parameters | .lparam | learning parameters: |
| dci parameters | .dparam | dci parameters. |
| hci parameters | .hparam | hci parameters. |
| scale parameters | .sparam | scaling parameters. |
| atest parameters | .atparam | atest parameters. |

The minimum required files are the filestem.names and filestem.dat files. These files describe the data being used as input and provide the examples required for learning. All other files are used for additional features of AQ17.

Tables generated by AQ17 are also saved in the filestem.suffix format. A description of each of these files is given below.

| Name | Suffix | Content |
|---|---|---|
| rules | .rule | rules generated from training data. Contents of file depend on setting of echo parameter in filestem.lparam |
| testing results | .test-results | results from testing rules in filestem.rule against testing data found in filestem.test. Contents of file are generated by ATEST module. |
| scaling intervals | .intv | results from SCALE module. List definitions of new attribute value intevals. Used to discretize testing data to same values. |
| tree | .tree | descision tree generated from rules by DT module. |
| hci log | .hlog | record of changes made to data by HCI module. |

### *3.3 Running AQ17*

If the current working directory contains the executable AQ17 file and the required input, then executing the following command, under the UNIX shell

**aq17 [option flags] filestem**

will start the aq17 process running with input from the files beginning with filestem in the current directory using the options described in the flags. These flags are used to invoke some the additional features of AQ17. All available flags and their meanings are given below:

| Flag | Meaning |
|---|---|
| **- d** | invoke the dci module for data-driven constructive induction (see section 5 for a full description of dci). If a filestem.dparam file is in the current directory, then this will be used to set the options for dci otherwise the default settings will be used. |
| **- e** | toggle expert mode on. With the -e option AQ17 checks the current directory for a filestem.lparam file. If this file exists it is used for setting the learning parameters. If the -e option is not used default settings will be used for the learning parameters. |
| **- h** | invoke the hci module for hypothesis-driven constructive induction (see section 6 for a full description of hci). If a filestem.hparam file is in the current directory, then this will be used to set the options for hci otherwise the default settings will be used. |
| **- n *th1*** | invoke the nt module for concept-driven noise removal. *th1* controls the degree of filtration perfomed. See section 7 for a detailed description of nt. |
| **- t** | invoke the testing module after rules have been learning. AQ17 will use the testing data found in filestem.test to test the rules generated from the training data in filestem.dat |
| **- s** | invoke the scaling module for $\chi^2$ based attribute-value discretization. AQ17 will take the data in filestem.train and automatically discretize it into significant intervals. Discretization is controlled using the parameters found in filestem.sparam. |

## 4. User's Guide

This section describes the tables in AQ17, their purpose, and syntax. Most tables are optional and many parts within a table are also optional.

### *4.1 The Title table*

The title table is an optional table useful for putting comments into the input file. This table is not used by AQ in any way. This table should be placed in the filestem.param file. A title table consists of two parts:

# (1..n)
A mandatory column which contains the row number of the text in the next column. Row numbers are consecutive integers beginning with "1".

text " "
The column consists of a string of characters surrounded by quotes. The string must fit on one line. Quotation marks may either by single or double.

### *4.2 The lparameters Table*

The mandatory parameters table contains values which control the execution of aq17. All of the parameters have default values. The default values are provided in parentheses following the description of each option. Each row of the parameters table represents one run of the program. In this way the user is allowed to specify many runs of the program on the same data in a single input file.

**run (1..n)**
An optional parameter which contains the row number of the text in the next column. Row numbers are consecutive integers beginning with "1".

**mode (ic)**
An optional parameter which controls the way in which AQ17 is to form rules. Legal values for this column are:

| | |
|---|---|
| ic: | "interescting covers" mode produces rules which may intersect over areas in the learning space in which there are no events. |
| dc: | "disjoint covers" mode produces rules that do not intersect at all. |
| vl: | "variable-valued logic" mode produces rules are are order dependent. That is, the rules for class n will assume that the rules for classes 1 to n-1 are not satisfied. Hence the rule for the last class will be null. |

**ambig (neg)**
An optional parameter which controls the way ambiguous examples are handled (i.e. overlapping examples from more than one class) are handled. Examples overlap when they have at least one common value for each variable. Legal values are:

| | |
|---|---|
| neg: | ambiguous examples are always taken as negative examples for the current class, and are therefore not covered by any classification rule. |
| pos: | ambiguous examples are always taken as positive examples for the current class, and are therefore covered by more than one classification rule. |
| empty: | ambiguous examples are ignored, i.e. treated as though they do not exist, and may or may not be covered by some classification rule(s). |

**trim (mini)**
An Optional parameter that specifies the generality of the resulting rules. The legal values are:

| | |
|---|---|
| gen: | rules are as general as possible, involving the minimum number of extended selectors, each with a maximum number of values. |
| mini: | rules are as simple as possible, involving the minimum number of extended seletors, each with a minimum of values. |
| spec: | rules are as specific as possible, involving the maximum number of extended seletors, each with a minimum of values. |

**wts (cpx)**
Optional parameter that specifies whether AQ is to display weights with the rules it produces. Legal values are:

| | |
|---|---|
| no: | no weights |
| cpx: | two weights are associated with each complex. The first weight is the total number of positive events that the complex covers (the *total* weight), The second weight is the number of events that this complex, and no other complex in the rule covers (the *unique* weight). Complexes are always displayed in decreasing order of the first weight. |

evt: in addition to the two weights (total and unique) calculated for each complex, a list of example indices is printed. These indices list the positive examples which are covered by the complex.

sel: weights are calculated for each selector in addition to each complex. There are two weights associated with each selector. The first weight is the number of positive examples covered by the selector, and the second weight is the number of negative examples covered by the selector. When selector weights are shown, selectors are displayed in decreasing order of the first weight. Otherwise selectors are displayed in the order they are given in the variables table.

all: all weights and example information is printed for each selector and complex. *all* is the union of evt and sel.

**maxstar (10)**
Optional parameter that specifies the number of alternative solutions kept during complex formation. A higher number specifies a wider beam search, which also requires more computer resources. In general the size of maxstar should be approximately the same as the number of variables used. The rules produced tend to be a good compromise between computational resources and rule quality. Maxstar values may range from 1 to 50.

**increment (1)**
Optional parameter that must be an integer and can take any value between 1 and 100. This parameter controls n, the number of subsets into which the training examples are divided. If n=1 then batch learning is done. If n>1 then incremental learning is performed, with the rules learned from examples subsets 1..n-1 being used to produce inhypos for the session in which the nth training set is used.

**echo (pvne)**
Specifies which tables are to be printed to output. Values in this column consist of a string of characters. Each character represents a single table type. The order of characters in the string controls the order of the tables in the output. No blanks or tabs are allowed in this string (such whitespace separates words in the input and would confuse the parser). Legal values for the echo parameter and the tables they represent are shown below:

0 ----- no echo
a ----- arules/lrules table
b ----- childrens table
c ----- criteria table
d ----- domaintypes table
e ----- events tables
i ------ inhypo table
n ----- names tables
p ----- parameters table
q ----- tevents tables
s ----- structure table
t ------ title table
v ----- variables table
z ----- learning time

**criteria (default)**
Entry is the name of the criteria table to be used. The name must be of alpha type, and a criteria table with that name must appear in the input file.

A sample table is shown below. Values in the first row are the default. Note that the default criteria table is the ONLY one for which it is unnecessary to follow with a full table description. The mincost criteria table, however, must be defined later in the input file. See next section for a description of -criteria tables. Parameters not present (e.g. "run", "mode" and "maxstar") take their default values.

Example:

```
parameters
ambig  trim   wts  echo  criteria  increment
neg    mini   cpx  pvne   default  1
pos    gen    all  pvne   mincost  20
```

### *4.3 The criteria tables*

All criteria tables other than the "default" *must* be defined. This table type is used to define a lexiographic evaluation function (LEF). An LEF evaluates a series of criteria in order, with the most important criterion being used first, and so on. Examples that fail to meet the first criterion are eliminated, while those that qualify are only then evaluated on the second criteria. The LEF is used by AQ15 to judge the quality of complex formed during learning. A LEF consists of several criterion-tolerance pairs. The ordering of the criteria in the LEF determines the relative importance of each. The tolerance specifies the allowable error within each criterion.

A criteria table name consists of two parts - the specific name, which must appear in the "criteria" column of the parameters tables (in the example above "mincost" was used) and the table name, -criteria. In the previous example "mincost" in the parameters table refers to the existence of a "mincost-criteria" table later in the input file. Any value in the criteria column of the parameters table except "default" must have a corresponding -criteria table and vice-versa.

**# (1..n)**
This column numbers the entries in the criteria table. Values must be sequential integers. This column is not required.

**criterion (maxnew, minsel)**
This mandatory column specifies the criteria which is to be applied at this point in the LEF. There are eight defined criteria. From these eight the LEF that best describes the user's rule preference is built. At least one and at most all eight criteria can be used in a criteria table. Criteria may also be selected by number rather than name. These numbers are the indices of the criterion in this table.

| | |
|---|---|
| maxnew(1)- | maximize the number of newly covered positive events, i.e. events that are not covered by previous complexes. |
| maxtot(2) - | maximize the total number of positive events covered. |
| newvsneg(3) - | maximize the ratio between the total number of newly covered positive examples and all negative evente covered. Computationally expensive. |
| totvsneg (4) - | maximize the ratio between the total number of positive covered examples and all negative evente covered. Computationally expensive. |
| mincost (5) - | minimize the total cost of the variables used (see section 4.4). |
| minsel (6) - | minimize the number of extended selectors. |
| maxsel (7) - | maximize the number of extended selectors. |
| minref (8) - | minimize the number of references in the extended selectors. |

**tolerance (0.00)**

This mandatory column specifies the relative tolerance in the importance of this criterion. In a strict LEF any complex not having the best (or equal) value for a criterion is immediately eliminated. This real value specifies the degree of tolerance in the importance of the criterion given in the same line. As an example, say the best complex in a list had a value of 100 for some criterion, and the tolerance for this criterion was 0.2, The absolute tolerance value is the product of the tolerance value (0.2) and the best value (100) and is 20. Any complex with a value between 80 and 100 will not be eliminated from the list.

An example is given below. The first -criteria table given is the default. In many experiments, this criteria table produces good results. This is the only -criteria table that need not be defined. The second example is a user-defined table called "mincost". Note that row numbers may be omitted.

Example:

```
default-criteria
# criterion tolerance
1 maxnew   0.00
2 minsel   0.00

mincost-criteria
criterion  tolerance
 mincost      0.20
 maxtot       0.00
```

### *4.4 The domaintypes table*

The domaintypes table is used to define domains for attributes. This table is optional, but it is convenient if several attributes have the same set of possible values. The table consists of four columns. The type, levels, and cost columns all have the same meanings as defined in the variables table description. There is no limit to the number of domains, or to the number of values (levels) in a domain.

**name ($x_n$)**
This mandatory column is the name of the domain being defined and must be of alpha type. If the name is not provided, the default name will be $x_n$ where $n$ is the index of the entry in the domaintypes table.

**type (nom)**
This optional column specifies the type of the domain being defined. Four domain types are legal. The legal types are described in the description of the variables table.

**levels (2)**
This optional integer value specifies the number of possible values for the domain being defined. There is no preset limit on the number different values or levels an attribute may have.

**cost (1.00)**
This optional real value specifies the relative expense of the domain being described. This value is used by the mincost criterion in the LEF (see description of criteria table). The expense of an attribute may be determined by the difficulty or expense of acquiring the value, or it may be set by a domain expert to encourage or discourage this attribute's appearance in generated rules.

The domaintypes table is normally used in conjunction with the variables table and the names table. Below are examples of both the variables table and domaintypes table.

Example:

domaintypes

| name | type | levels |
|---|---|---|
| boolean | nom | 2 |
| op_system | nom | 2 |
| floppies | lin | 4 |
| processor | nom | 3 |
| memory | str | 8 |

variables

| # | name | cost |
|---|---|---|
| 1 | pascal.boolean | 10.0 |
| 2 | fortran.boolean | 10.0 |
| 3 | cobol.boolean | 10.0 |
| 4 | op_system | 10.0 |
| 5 | floppies | 100.0 |
| 6 | disk.boolean | 0.0 |
| 7 | processor | 1.0 |
| 8 | memory | 100.0 |
| 9 | printer.boolean | 0 |

### *4.5 The variables table*

The mandatory variables table specifies the names and domains (legal values) of the variables used to describe events. The variables table must include at least one, and at most all 5 of the following columns. There is no preset limit to the number of variables, or to the number of values (levels) of a variable.

**# (1..n)**
This optional column numbers the entries in the variables table. Values must be sequential integers.

**name (xn)**
This column specifies the name of the attribute. Names must be of alpha type. If the name column is omitted, the default value is x*n* where *n* is the number of the row in which the name.domain-name. Name is the alpha string name of the specific variable while domain-name is a more general name of the domain (as defined in the domaintypes table). The name and domain-name may be the same if there is only one variable with that domaintype.

**type (nom)**
This column specifies the type of the variable domain. Four domain types are legal:

| | |
|---|---|
| nom | a "nominal" domain consists of discrete, unordered values (e.g. colors) |
| lin - | a "linear" domain consists of discrete, ordered alpha or numeric values (e.g. sizes – small, med, large) |
| cyc- | a "cyclic" domain consists of discrete values in a circular order (e.g. months). |
| str - | a "structured" domain has values in the form of a hierarchical taxonomy. A variable with a structured domain requires that domain be described in a structure table as well. |

**levels (2)**
This integer gives the size of the domain being defined. Their is no present limit on the number of values an attribute may have.

**cost (1.00)**
This real number specifies the relative 'expense' of the domain being defined on this line. The value is used when computing criterion mincost is used in the LEF. Cost defaults to 1.0

The variables table may be used in conjunction with the domaintypes and -names tables. In the example given above, the type and levels columns were defined for all domaintypes so that these columns were not needed in the variables table.

### *4.6 The names tables*

The names table is used to specify legal domain values for an attribute. These must be the attribute values that appear in the events tables. If no names table is present in the input file, then the values for that domain are assumed to be the integers from 0 to #levels-1 (levels are defined in the variables table). The specific name of a names table must be the same as that used in the domains or variables table. There are two required columns in each names able: value and name.

**value (1..n)**
This column must be an integer beginning with '0' and continuing sequentially up to #levels-1. This column is the integer equivalent of the name to be defined in the next column. This column is required. There is no preset limit to the number of values for an domain being defined.

**name (1..n)**
This column defines the input and output name of the value being defined. Alpha, integer or real types are allowed. Only two decimal places are stored for real types. This column is required.

Below are typical examples of the -names table. All variables that are of type "boolean" may take values "yes" and "no". The domain "make" has the value "IBM", "Compaq", "Zenith" and "Apple". Note that for the variable "floppies" the default values of 0,1,2 and 3 are acceptable so there is no need for a "floppies" names table.

Example:

boolean-names

| value | name |
|---|---|
| 0 | no |
| 1 | yes |

make-names

| value | name |
|---|---|
| 0 | IBM |
| 1 | Compaq |
| 2 | Zenith |
| 3 | Apple |

### *4.7 The structure tables*

The structure table is optional and is used to define a structured domains for any variable of the structured type (as specified in the domaintypes or variables table). A structured domain has the form of a hierarchical graph, where the lowest level corresponds to the values of the variable as they will appear in the input examples (and may be defined in a names table.) Higher levels (as definedin the structure table) specify parent nodes in the hierarchy of values and are used to simplify classification rules.

The specific name of a structure table must be the name of the domain, as specified in the name column of the domaintypes table. If the domaintypes table is not specified it may be a variable name from the variables table. A structure table consists of three columns:

**name**

This optional alpha or integer type entry specifies the name of the corresponding value in the tree.If specified, it may appear in classification rules instead of the values named in the names table, or events-table.

**value**

This mandatory integer entry specifies the node in the hierarchy which is the parent of the nodes specifies in the subvalues column. If this value is a subvalue of some other values it must appear before any rows inwhich it is listed as a subvalue. This value must always be greater than any of the subvalues in the following columns.

**subvalues**

This mandatory entry specifies a set of children values for the parent node as defined in the value column. This entry consists of a string of integers separated by commas or by ".." as in extended selectors. These numbers correspond to values as defined in the names table of the variable or previous rows of the structure table.

The hierarchical graph below shows an example of a structured domains for the variable "memory". Note that the same node (e.g. 64) may be shared by multiple parents ("medium" and "large").

For the variable "memory" a names table must first be defined, because to allow a domain of all values between 2 and 280 would be inefficient and might cause inaccurate rules. Note that the node "large" must be defined after the node "very_large" as it is higher in the tree.

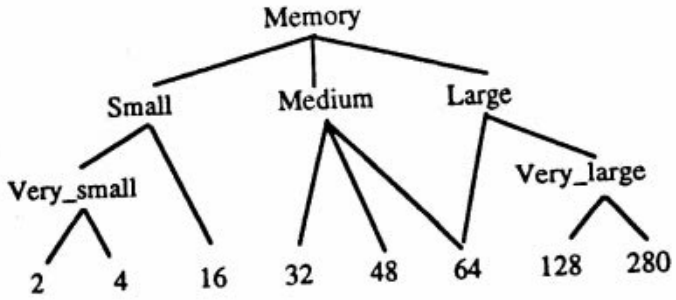
Example:

memory-names

| value | name |
|---|---|
| 0 | 2 |
| 1 | 4 |
| 2 | 16 |
| 3 | 32 |
| 4 | 48 |
| 5 | 64 |
| 6 | 128 |
| 7 | 280 |

memory-structure

| name | value | subvalues |
|---|---|---|
| very_small | 8 | 0,1 |
| small | 9 | 8,2 |
| medium | 10 | 3..5 |
| very_large | 11 | 6,7 |
| large | 12 | 5,11 |




### *4.8 The inhypo tables*

The inhypo tables are optional and are used to input rules for incremental learning. The specific name of this table must match the name of one of the decision classes. The rules input in the inhypo table have two possible roles. In the first, when there is at least one events table specified, the input rules are used as initial covers for incremental learning. If no events tables are present the inhypo rules are treated as events for rule optimization. If inhypo rules intersect, then their intersection is treated as determined by the "ambig" parameter (see parameters table). There are two columns in the inhypo table.

**# (1..n)**

This mandatory column associates a number with each complex in the rule. It is a sequentially increasing integer (1..#complexes). Only in inhypo tables may an entry span more than one line. There must always be a # entry for each complex in the table.

**cpx ([])**

This mandatory column specifies the $VL_1$ rule. A complex is presented as a conjunction of selectors. Selectors, and complexes are defined in section 2.

Below are examples of an inhypo tables. A complex is a conjunction of selectors and a cover is a disjunction of complexes.

Example:

Under1000-inhypo
 # cpx
 1 [floppies=0]

Fron1000_4000-inhypo
 # cpx
 1 [Floppies = 1,2][memory>16]
 2 [Floppies > 3][memory<=4]

### *4.9 The data tables*

Data tables for AQ17 can take one of two forms: 1) a single table of attribute values with individual examples listed as rows in which the value of the class variable is given in the last column or 2) multiple tables of attributes in which all examples with the same class variable value is grouped together. The first format is the same as that used by C4.5 and ID3. The latter format is the -event table format used by AQ15. Both format types are supported in order to make using AQ on a given problem as easy as possible. The first format will be referred to as the single-table format, while the other will be the multi-table format.

The single table format is very simple. The input data consists of a header followed by n rows of attribute-value vectors (examples). The header is a list of attribute names such as x1,x2,...xn. The last column is the class variable. Attribute values may be separated by commas, or spaces. Attribute values may be alphnumeric strings, or real, or integer -values. If data is provided in single-table format, AQ17 will automatically build the data into multi-table format. AQ17 detects the need to build data tables when it cannot find any filestem.tdat (table data) files. AQ17 then builds the filestem.tdat files and invokes AQ.

If data is already in a multi-table format then it should be stored in filestem.tdat. The multi-table format consists of a set of -events (or -tevents in the case of testing examples) tables. The format of -events (and -tevents) is provided below.

Events and tevents tables have identical structure except the examples contained in events tables are used for learning and those in tevents are used in testing. Events tables contain a specific name corresponding to the name of the decision class. This name must be of type alpha.

The column headers for this table consist of the attribute names (as defined in the variables table). The values in the row of the table must be legal values for the appropriate attribute. In the case that many attributes are used, events tables may be split. Each split table must contain the specific name and 'events' and a different set of attributes. Attributes can not overlap between split events tables. Events tables consists of two column types: 1) row number and 2) attribute name.

**# (1..n)**

This optional column is an integer index of the example. Values must be sequential integers beginning with 1. This column is optional.

**variables (x1..xn)**

This column consists of an arbitrary number of columns, one for each attribute in the variables table. The entries in the rows of the table must contain legal values of the corresponding variables in the heading. Entries may be single values or they may be an 'unknown' symbol (*). Unknown values (*) are internally represented as taking all legal values for that attribute domain. Below is an example of events tables.

Example:

Under1000-events

| Pascal | Fortran | Cobol | floppies | Disk | Processor | Memory | Printer |
|---|---|---|---|---|---|---|---|
| no | no | no | 0 | no | M6502 | 2..16 | no |
| no | no | no | 0 | no | Z80 | 32 | no |

Over4000-events

```
# Pascal  Fortran  Cobol
1 yes     yes     yes
2 no      yes     yes

Over4000-events
 #  floppies Disk  Processor Memory  Printer
 1  1        yes   Z80       128     no
 2  2        no    I8085     64      yes
```

### *4.10 The children tables*

The optional children tables define the hierarchical ordering of the decision variable. The specific name of the table must be the name of a class already defined, i.e. the name must have appeared as the specific title in an events table. The rule base may be structured to arbitrary depth. The children table consists of two columns:

**node**

This mandatory alpha column specifies the name of the node being defined.

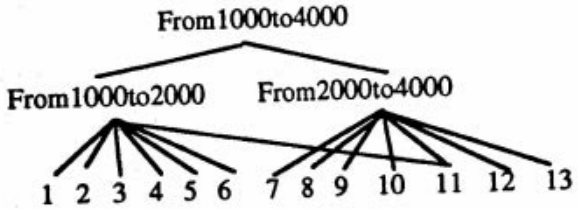
**events**

This mandatory column is a list of event indices belonging to the parent node which are examples of this child node. The list of indicies may use commas (1,2,3,4) or ranges (1..4). The parent's events are numbered in the order they appear in the events table. All indicies refer to the example indices of the root node of the tree so that all events use the same set of numbers.

The tree below shows how a decision attribute may be structured. In this case classes "Under1000", "From1000to4000" and "Over4000" are brothers at the top of the structure. The class "From1000to4000" has two sub-classes, "From 1000to2000" and "From2000to4000".

The example below defines the two classes ""From1000to2000" and "From2000to4000" which are sub-classes of the class "From1000to4000". Assuming that there is already an events table for "From1000to4000" with 13 events, the following children tables would assign events 1 to 6 and 11 to class "From1000to2000" and events 7 to 13 to class "From2000to4000".

```
From1000to4000-children
  node            events
  From1000to2000  1..6,11
  From2000to4000  7..13
```



## 5. Data-Driven Constructive Induction

### *5.1 Introduction*

Data-driven constructive induction is a method for generating new problem-relevant attributes based on an analysis of the training data. This generation is performed through the application of various mathematical and logical operators to the initial attribute values. The currently available operators include multiplication, integer division, addition, subtraction, comparison, average, most-common, least-common, maximum, minimum, and number of attributes having some value. The DCI method can be invoked with the -d option and and requires a filestem.dparam file be present in the current directory. The following sections will describe the tables which should be included in the dparam file.

### *5.2 The DCI Method*

The DCI method for constructing new attributes is primarily one of generate and test. This method is shown in figure 2a. First all candidate features are identified (all linear-type features). Then each pairwise combination of attributes and operators is calculated. The operation to be performed on this pair is selected from the list supplied by the user. With the features, and operation selected the values for the new feature are calculated. The discriminatory power of these feature values is then tested using an information-theoretic metric. This is the same metric used in ID3 to determine which attribute to select next when building a decision tree (Quinlan, 1983). A brief description is included here for completeness.

In this approach the quality of an attribute is a measure of how well the values of the attribute discriminate objects of different classes (how much information is provided by the values of that attribute). If the probability of the object belonging to a class is $p^+$ and if the probability of the object belonging to another class is $p^-$ then the information content of a message is:

$$\text{Info(message)} = -(p^+ \log_2 p^+) - (p^- \log_2 p^-)$$

With a known set of objects the probabilities can be approximated by relative frequencies; $p^+$ is the percentage of objects in C with class "+".

The set of all attributes has a total information value. This is denoted M(c). This is the total information in the message. Each individual attribute contributes to that information value. The information value of an attribute is the sum of the information values of each of the attribute's values.

Attribute quality is calculated for each new attribute. If the information value of a newly constructed attribute is above a user-defined threshold, the attribute is retained, if not, the attribute is discarded.

A number of different operations are available to construct new features. These operations can be classified as either binary operators or multi-argument operators (functions). In the binary group are currently the comparison operator, and a number of mathematical operators including addition, subtraction (absolute difference), multiplication, and integer division. Integer division is calculated as trunc(x/y). Examples of each of these operations are shown below:

| **Operator** | **feature 1** | **feature 2** | **result** |
|---|---|---|---|
| comparison | 6 | 8 | 2 * |
| addition | 6 | 8 | 14 |

| | | | |
|---|---|---|---|
| subtraction | 6 | 8 | 2 |
| multiplication | 6 | 8 | 48 |
| integer division | 6 | 8 | 0 |

*The comparison operator can take three different values depending on the relation between the value of attribute x and attribute y. if x=y then compare(x,y)=1; 2 if x < y; and 3 if x > y

In the multi-argument class are the following functions: maximum, minimum, average, least_common, most_common, and #VarEQ(x). Except for the latter, these function are self-explanatory. #VarEQ(x) is a function which calculates the number of times the value x appears in an example. For a vector of binary attributes, #VarEQ(1) measures the number of variables (attributes) that take the value 1 in an example of a given class. Tiebreakers in the calculation of functional operators is done on the basis of which value appeared first in the example (from left to right). Examples of each of these operations is shown below:

| **Operator** | **feature1** | **feature2** | **feature3** | **feature4** | **result** |
|---|---|---|---|---|---|
| maximum | 4 | 8 | 6 | 6 | 8 |
| minimum | 4 | 8 | 6 | 6 | 4 |
| average | 4 | 8 | 6 | 6 | 6 |
| most_common | 4 | 8 | 6 | 6 | 6 |
| least_common | 4 | 8 | 6 | 6 | 4 |
| #VarEQ(4) | 4 | 8 | 6 | 6 | 1 |
| #VarEQ(6) | 4 | 8 | 6 | 6 | 2 |
| #VarEQ(8) | 4 | 8 | 6 | 6 | 1 |

The program has a default list of global functions, but allows the user to modify the list to fit the problem at hand. The default list of functions include maximum, minimum, average, most frequent, least frequent and #VarEQ(x).

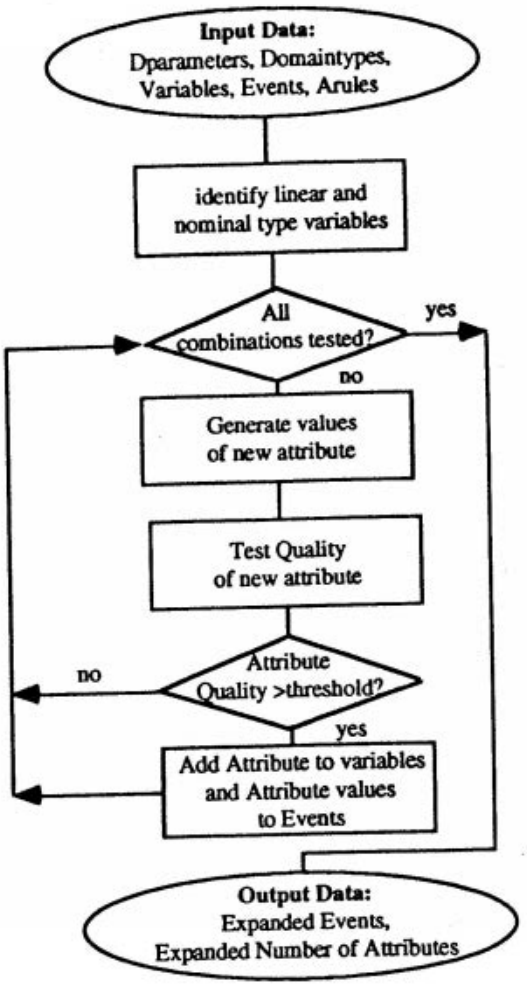



*Figure 2a.* The DCI Method

### *5.3 DCI Tables*

#### 5.3.1 The dparameters table

The mandatory dparameters table contains values which control the execution of the DCI module.

**maxnew (10)**
Maxnew is an optional integer that specifies the maximum number of new attributes that will be kept. As soon as the number of new attributes exceeds maxnew, no new attributes are added to the current attribute set. The default value of maxnew is 10.

**operations (default)**
Entry is the name of the operations table to be used. The name must be of alpha type, and an operations table with that name must appear in the input file.

An example dparameters table is shown below. These parameters state that only at most will five new attributes be constructed (maxnew=5), and the operations table is default. The default operations table is the ONLY one for which it is unnecessary to follow later with an operations table.

Example

parameters
operations maxnew
default 5

### 5.3.2 The operations table

The operations table defines the list of operations the user wishes to perform on the data. With each operation entry is included the quality threshold that new attributes constructed using that operator must attain, as is a maximum cost threshold for which the cost of the new attribute must not exceed.

Each operations table title line has two parts - the specific user-defined name, and the keyword 'operations'. There may be multiple operations tables in the input. The table that is used must have its user-defined name listed in the dparameter table under operations. If the default operations table is desired only the word 'default' in the dparameters table is required. The first operations table shown below is the default. The second is user - defined.

default-operations

| operation | threshold | cost |
|---|---|---|
| addition | 0.1 | 15.0 |
| subtraction | 0.1 | 15.0 |
| mult | 0.1 | 15.0 |
| relation | 0.1 | 15.0 |
| valuecom | 0.1 | 15.0 |

numeric-operations

| operation | threshold | cost |
|---|---|---|
| addition | 0.5 | 10.0 |
| subtraction | 0.6 | 10.0 |
| mult | 0.7 | 10.0 |
| division | 0.7 | 10.0 |

## 6. Hypothesis-Driven Constructive Induction

### *6.1 Introduction*

The hypothesis-driven constructive induction (HCI) method incrementally transforms the representation space by analyzing inductive hypotheses generated in one iteration and using detected patterns as attributes for the next iteration. The proposed method is based on repetitively detecting strong "patterns" in the hypotheses generated in one iteration, and then treating them as new attributes in the next iteration. To explain the method, we will start by describing the measure of a pattern's strength.

### *6.2 Determining the Pattern Strength*

A pattern can be a group of rules in the learned description, a part of a rule (a conjunction of conditions), or a group of attribute values in a condition of a rule. The strength of a pattern can be determined in many different ways. Below is the measure implemented in the HCI module.

In this measure, the strength ($\sigma$) of a pattern is a function of the number of positive examples, PCov, and the negative examples, NCov, that are "covered" by the pattern:

$$\sigma(pattern) = f(PCov(pattern), NCov(pattern)) \quad (1)$$

To determine the specific form of the function f, let us observe that the strength of a pattern should be positively related to the number of positive examples covered by it, and negatively related to the number of negative examples covered by it. The way the strength is calculated should also depend on the pattern type—is it a ruleset, a subrule, or a group of attribute values. In addition, the measure of strength may distinguish between types of coverage of concept instances by a given pattern. For example, a concept instance can be covered only by a given pattern (unique coverage), or it can be multiply covered. To reflect this difference, PCov and NCov are expressed not just by single numbers, but by multiple numbers. Here is a simple measure of pattern strength that reflects above considerations:

$$\sigma(pattern) = \frac{t^+(pattern) + \lambda u^+(pattern)}{t^-(pattern) + 1} \quad (2)$$

where

$t^+$(pattern), called the *total positive weight*, and $t^-$(pattern), called the *total negative weight*, are the numbers of positive and negative examples covered by the pattern, respectively.

$u^+$(pattern), called the *unique weight*, is the number of positive examples uniquely covered by the pattern, i.e., not covered by any other comparable pattern.

$\lambda$ is a parameter that controls the relative importance given to these two types of coverage.

When $\lambda = 0$, i.e. when the unique weight is ignored, the above measure of pattern strength ($\sigma$) is similar to the *logical sufficiency* (LS) used in the Prospector expert system (Duda, Gasching and Hart, 1979), and in the STAGGER concept learning system (Schlimmer, 1987).

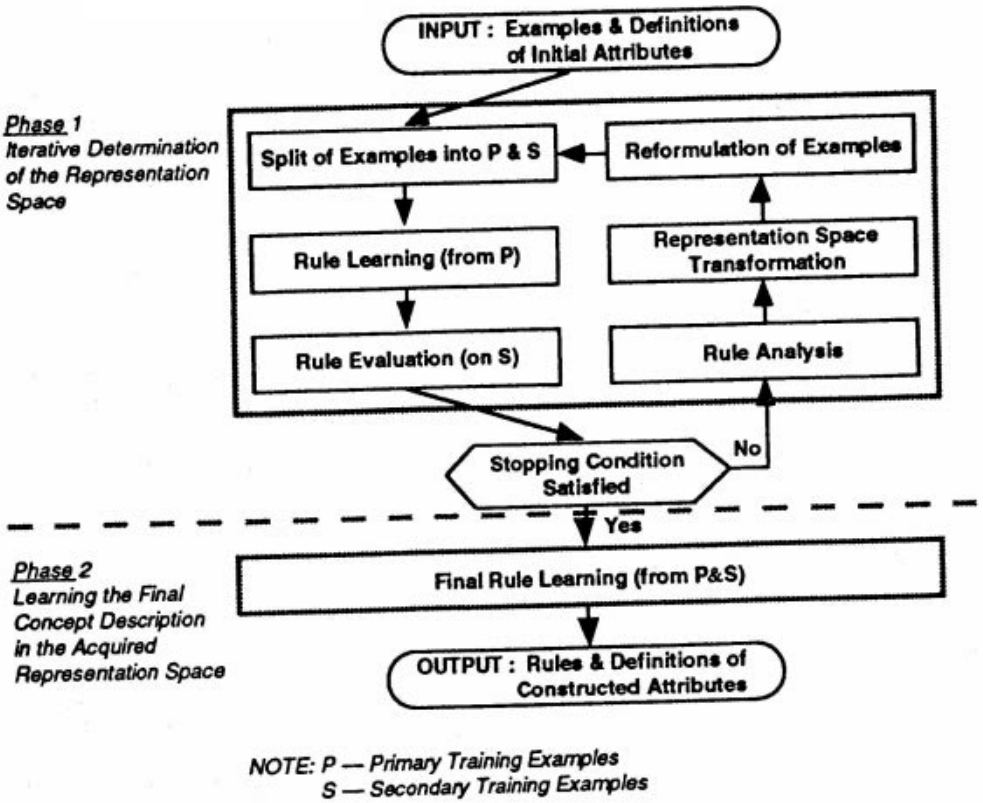



*Figure 2b.* The HCI method.

### *6.3 Determining an Admissible Ruleset*

The HCI method works iteratively. Each iteration generates a complete and consistent set of rules, i.e., a ruleset that covers all positive examples and none of the negative examples. In order to speed up the process of determining strong patterns, and avoid searching through rules that are weak and/or low validity, the method selects rules that have sufficient strength in the generated ruleset. These rules constitute an *admissible ruleset*.. The method searches for strong patterns in the admissible ruleset, uses patterns for transforming the representation space, and then moves to the next iteration (the complete method is described in the next section).

When determining the strength of rules in a ruleset representing a concept, expression (2) can be simplified; specifically, the denominator in (2) can be ignored. This is so because such a ruleset is consistent with regard to negative examples (no negative examples covered), and therefore $r$ (the negative weight ) is zero.

Thus, we have:

$$\sigma(rule) = t(rule) + \lambda u(rule) \qquad (3)$$

where, $t$ is the total (positive) weight of a rule in a ruleset (the total number of training examples covered by this rule); and $u$ is the unique weight of a rule in a ruleset (the number of training examples covered only by this rule, and not by any other rule in the ruleset; Michalski et al.,

1986). The program's default value for parameter $\lambda$ is 2, which gives a relatively strong preference to rules with higher unique weights, i.e., rules that have smaller overlap with other rules in a ruleset for a given concept.

To determine an admissible ruleset, rules in the ruleset for a given concept are ordered from the strongest to the weakest. An *admissible* ruleset contains the minimal number of rules from the ruleset whose total relative strength exceeds a predefined threshold:

$$\frac{\sum_{i=1..m} \sigma_i}{\sum_{j=1..n} \sigma_j} \geq TH \qquad (4)$$

where $\sigma_i$ is the strength of rule (i) defined by equation (3), n is the total number of rules in the current hypothesis, m ($m \leq n$) is the number of strongest rules (recall that the rules are ordered, thus $\sigma_i \geq \sigma_{i+1}$). In the program, the TH parameter has the default value 0.67, which means that the admissible ruleset will cover at least 2/3 of the training examples of a given concept. Since noisy examples and exceptions are normally covered by low-strength rules, therefore the admissible ruleset can be expected to cover the most "central" portion of the learned concept. This method could be improved by setting the TH parameter on the basis of knowledge of the noise level, and of the confidence in the learned hypothesis.

### *6.4 The HCI Method*

The proposed HCI method combines an inductive rule learning algorithm ($A^q$) with a procedure for iteratively transforming representation space. In each iteration, the method changes the representation space by adding new attributes, and/or removing insufficiently relevant attributes. The quality of the hypothesis generated in each iteration is evaluated by applying the hypothesis to a subset of training examples. The set of training examples prepared for a given iteration is split into the primary set (the P set), which is used for generating hypotheses, and the secondary set (the S set), which is used for evaluating the prediction accuracy of the generated hypotheses. Figure 2b presents a diagram illustrating the HCI method.

For the HCI module input consists of training examples of one or more concepts, and background knowledge about the attributes used in the examples (which specifies their types and legal value sets). For the sake of simplicity, let us assume that the input consists of positive examples, $E^+$ and negative examples, $E^-$, of only one concept. If there are several concepts to learn, examples of each concept are taken as positive examples of that concept, and the set-theoretical union of examples of other concepts is taken as negative examples of that concept.

The method consists of two phases. Phase 1 determines the representation space by a process of iterative refinement. In each iteration, the method prepares training examples, creates rules, evaluates their performance, modifies the representation space, and then projects the training examples into the new space. This phase is executed until the *Stopping Condition* is satisfied. This condition requires that the prediction accuracy of the learned concept descriptions exceeds a predefined threshold, or there is no improvement of the accuracy over the previous iteration. Phase 2 determines final concept descriptions in the acquired representation space from the complete set of training examples. The output consists of concept descriptions, and definitions of attributes constructed in Phase 1. Below is a detailed description of both phases, and of the basic modules of the method.

Phase 1 consists of six modules. The first, "Split of Examples" module, divides positive and negative training examples into the primary set, P, and the secondary set, S (in the experiments the

split was according to the ratios 2/3 and 1/3, respectively). The set of primary positive (negative) examples is denoted $P^+$ ($P^-$), and the set of secondary positive (negative) examples is denoted $S^+$ ($S^-$). Thus $P = P^+ \cup P^-$, and $S = S^+ \cup S^-$. The primary training set, P, is used for initial rule learning, the secondary set, S, for an evaluation of intermediate rules, and total set, $P \cup S$, is used for the final rule learning (in Phase 2).

The "Rule Learning" module induces a set of decision rules for discriminating $P^+$ from $P^-$, i.e., a cover COV ($P^+/P^-$) of positive primary examples against negative primary examples. This is done by employing the AQ15 inductive learning program (Michalski et al., 1986). The program is based on the algorithm $A^q$ for solving general covering problem, which was described in various sources, e.g., (Michalski and McCormick, 1971; Michalski, 1973).

The "Rule Evaluation" module estimates the prediction accuracy of the rules by applying them to the secondary training set, S. The accuracy of the rules in classifying the examples from S is determined by the ATEST procedure implemented in the AQ15 program (Reinke, 1984). If the Stopping Condition criterion is not satisfied, the control passes to the "Rule Analysis" module, otherwise, it passes to Phase 2.

The "Rule Analysis" module determines an admissible ruleset. The "Representation Space Transformation" module analyzes the rules in this ruleset to determine desirable changes in the representation space. It removes redundant or insignificant attributes, modifies existing attributes (by attribute value agglomeration), and generates new attributes. The "Example Reformulation" module projects all training examples into the new representation space, and the whole inductive process is repeated.

Phase 2 determines final ruleset by applying the "Rule Learning" module to all training examples projected into the final representation space determined in Phase 1. For each concept, a set of the most specific (ms) rules is induced from all positive examples against all negative examples, i.e., a cover $COV_{ms}$ ($E^+/E^-$), and the most general (mg) rules of negative examples against positive examples, that is $COV_{mg}$ ($E^-/E^+$). The final concept description is built by generalizing the most specific rules for positive examples against the most general rules for negative examples, i.e., determining a cover, $COV_{mg}[COV_{ms}(E^+/E^-)/COV_{mg}(E^-/E^+)]$ (notice that the arguments for the covering algorithm are here not sets of examples, but sets of rules). The description so generated represents an intermediate degree of generalization between the most *specific* positive rules and the most *general* negative rules. For details on generating such concept descriptions see Wnek (1993).

### *6.5 HCI Tables*

#### The hparameters table

The hparameters table is an extended version of the parameters table used to control learning. The HCI module requires both the learning parameters and hci parameters. This section will only describe the hci parameters (expand, window, th1, and th3). Section 4.2 describes the learning parameters.

#### expand (all)

Controls generation of new attributes. With expand=no – no attributes are generated; expand=one – attributes are generated only for one class; expand=all – attributes are generated for all classes.

**window (0.67)**

Controls the size of the primary training set (P). The default setting assumes 2/3 of the training set to be used as the primary training set, and 1/3 of the training set as the secondary training set.

**th1 (0.67)**

Threshold used for determining the admissible ruleset, th1 = TH in formula (4). In the program, the parameter has the default value 0.67, which means that the admissible ruleset will cover at least 2/3 of the training examples of a given concept. Since noisy examples and exceptions are normally covered by low-strength rules, therefore the admissible ruleset can be expected to cover the most "central" portion of the learned concept. For th1=1, all rules will be regarded as admissible.

**th3 (0.1)**

Threshold used for controlling removal of irrelevant attributes. The relevance of an attribute is measured as a ratio between its *importance score* and the *sum of importance scores of all attributes*. If relevance is less then th3, then the attribute is regarded as irrelevant. For each attribute, its importance score is calculated by summing up the *total* weights of all rules in which the attribute was used. The greater th3 is, the more attributes will be removed. For th3=0, no attributes will be removed.

## 7. Handling Noisy Data

### *7.1 Introduction*

Inductive learning systems must perform some form of generalization in order to anticipate unseen examples. Ideally, a concept description generated by an inductive learning system should cover all examples (including unseen examples) of the concept (completeness) and no examples of other concepts (consistency). This is precisely what most inductive learning systems do, generating a complete and consistent concept description. In the case of noisy data, complete and consistent descriptions are problematic because multiple concept descriptions can partially overlap in the attribute space. This is so because attribute noise skews the distribution of attribute value from the correct value. Because of the existence of noise in the sensory data, some positive examples are noise, that is, they are actually negative examples. We call such examples "positive noisy examples". These positive noisy examples are covered by the complete and consistent description. These examples, however, should not be covered. On the other hand, some negative examples can be noise, i.e., they are actually positive examples. Such negative examples are referred to as "negative noisy examples". Negative noisy examples are incorrectly left uncovered.

There are two basic approaches to learning from noisy data. One is to allow a certain degree of inconsistent classification of training examples so that the descriptions will be general enough to describe basic characteristics of a concept. This approach has been taken by the ID family of algorithms [Quinlan, 1986]. The second approach, used by programs which induce rules rather than trees, is to discard some of the unimportant rules and retain those covering the largest number of examples. The remaining rules are a general description of the concept. Typical algorithms using these techniques are the AQ family of algorithms [Michalski, 1986]. Rule truncation in AQ15 ([Michalski, 1986, Zhang and Michalski, 1989, Pachowicz and Bala, 1991], [Michalski, 1986, [Zhang and Michalski, 1989], and the significance test in CN2 [Clark and Niblett, 1989] are also examples of that approach. Other approaches are based on the minimum description length principle [Quinlan, 1989] and cross validation to control over-fitting during a training phase [Breiman, Friedman et al., 1984].

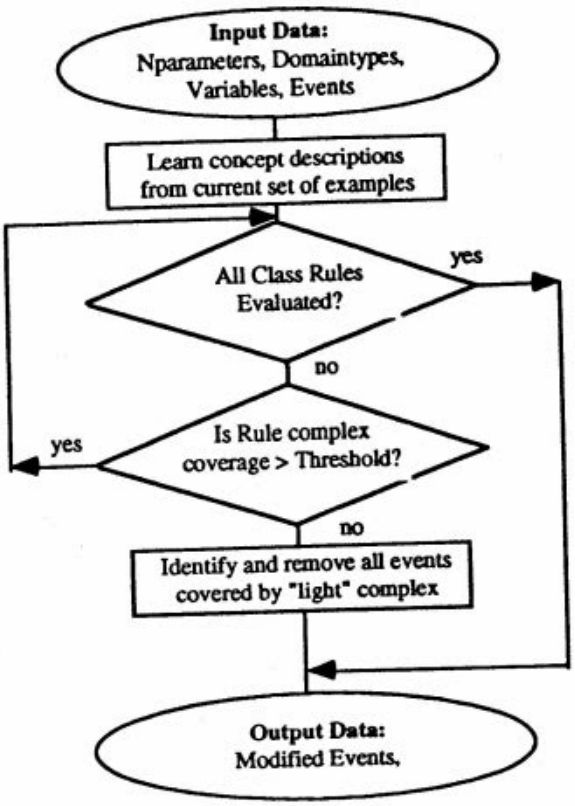



*Figure 2c.* The NT Method

Because the above methods try to remove noise in one step, they share a common problem - the final descriptions are based on the initial noisy data. For methods applying pre-pruning, the attributes used to split the instance space in a pruned tree are selected from initial noisy training data. For methods applying pre-truncation, the search for the best disjunct is also influenced by noise in training data. This problem is more severe for post-pruning and post-truncation. The post learning concept optimization cannot merge broken (by noisy examples) concept components (i.e., disjuncts, subtrees). So, the "gaps" in concept descriptions remain unfilled. Post learning optimization will cause the complexity of concept descriptions to decrease only if the concept components are eliminated; i.e., without reorganizing concept descriptions.

### *7.2 The NT Method*

A new approach to noise-tolerant learning, called model-driven learning, is presented. The approach was developed to acquire concept descriptions of visual objects from the teacher- or system-provided noisy training samples. This method has the following steps (Figure 2d): (i) learning concept descriptions, (ii) the evaluation of learned class descriptions and the detection of less significant disjuncts which do not likely represent patterns in the training data, (iii) the iterative removal (e.g., truncation or pruning) of detected disjuncts/subtrees, and (iv) the filtration of training data by optimized rules (i.e., removal of all examples not covered by truncated of pruned concept description). The first novel aspect of this approach is that rules optimized through disjunct removal are used to filter noisy examples, and then the filtered set of training data is used to relearn

improved rules. The second novel aspect is that noise detection is done on the higher level (model level) and can be more effective than traditional data filtration applied on the input level only. The expected effect of such a learning approach is the improvement of recognition performance and a decrease in the complexity of learned class descriptions.

### *7.3 NT Tables*

The NT module requires only one parameter which is given at the command line when AQ17 is invoked. For example,

> aq17 -n 5 filestem

sets th1 to 5 and invokes the nt module on filestem.dat. A full description of th1 is given below.

**th1**

This mandatory parameter controls the degree of filtration performed by NT. Using the example coveratge (t-weight) calculated for each learned rule, rules are ranked. Rules with low t-weight are 'light' rules and identify potentially noisy examples. Light rules are added to a ordered list of 'noise covering' rules (ranked in order of increasing weight) while the total t-weight of such rules is below th1*#examples. Values for th1 range from 0 to 100.

Example

th1=5; total examples=100: 50 examples of class1, 50 examples of class2
nparameters
th1
class1-outhypo
# cpx
1 [x1 = 11,14] [x2 = red] (t:40, u:47)
2 [x1 = 10][x2 = yellow] ( t:7, u:7)
3 [x1 = 15][x2 = red] (t:3, u:7)

In the case, complex #3 is put in the 'noisy set' and is then used to remove examples from the training data. Complex #2 is not used to filter as the t-weight total of complex #1 and complex #2 (10) would be greater than th1*#examples (5)

## 8. Handling Continuous Data

### *8.1 Introduction*

The AQ learning algorithm is designed to learn from attributes that have a small number of discrete values. This makes it well-suited to symbolic processing. However, their are many potential applications in which learning descriptions of classes involves data that has continuous data. For example in the case of texture recognition, feature extraction techniques such as Law's masks output real-valued data (to the hundredths place) over a wide range. The precision of these values is not required to do the learning, and in fact can overwhelm the learning method. The problem that SCALE addresses is to determine how to partition the data into meaningful intervals.

### *8.2 The SCALE Method*

The ChiMerge method (Kerber, 1992) has been implemented which discretizes attribute values into relevant intervals. The effect of this discretization is to remove attribute values from the data.

The ChiMerge algorithm is a bottom-up process in which initially all values are stored in separate intervals which are then merged until a termination condition is met. The interval merging process consists of continuously repeated two steps: compute the $\chi^2$ value (correlation between the value of the class attribute and the interval in which the example value belongs) and 2) merge the pair of adjacent intervals with the lowest $\chi^2$ value. All intervals are merged until all pairs of intervals have $\chi^2$ values exceeding the user defined chi-threshold. The chi-threshold is calculated from a table given a desired signifigance level and the number of degrees of freedom (1 less than the number of classes). The $\chi^2$ value measures the probability that the attribute interval and class value are independent. If means that among cases where the class and attribute are independent, there is a signifigance-level probability that the computed $\chi^2$ value will be less than the threshold value. If the interval has a $\chi^2$ value greather than threshold then class and attribute are not independent. Higher values of the threshold causes more merging and fewer intervals and vice versa. Testing of this method with AQ in the texture domain has shown it to improve recognition over simple equal-interval scaling.

The Chimerge technique can be used whenever there exists attributes in the problem being studied which have large domains. Usually this large number of values is not necessary-there is too much detail. In such cases rules can be overly complex and inaccurate. With the Chimerge technique the domains are reduced to only intervals which are dependent on the class.

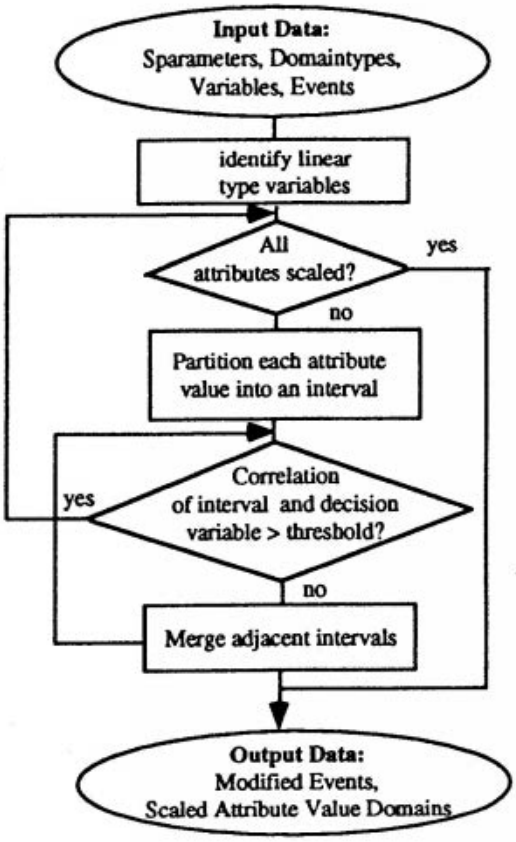



*Figure 2d.* The SCALE Method

### *8.3 SCALE tables*

SCALE requires only one table, the sparameters table. This table has 3 parts: method, intervals-output, and variables description.

**intervals (1)**
This parameter controls whether the resulting interval ranges determined by the method will be output with the scaled data. This information is required if testing data must be scaled into the same intervals into which the training data was scaled. This information may also be of interest to a domain expert who wishes to evaluate the semantics of the resulting intervals.

**variables**
The mandatory variables description provides information on the name, minimum number of intervals, maximum number of intervals, method of scaling and threshold values. The name must be an alpha string. The minimum and maximum columns specify the lower and upper bounds on how many intervals the scaling program is allowed to produce for the attribute being described. The method value controls what method is used to do the discretization. The legal values are chi and equal (for equal interval sized). The threshold value specifies how strong the correlation between decision variable and the interval must be to prevent it from being merged with its adjacent

interval. The signifigance of this correlation is obtained by looking up in a $\chi^2$ table the degrees of freedom of the data (#classes -1).

## 9. Decision Trees from Rules

### *9.1. Introduction*

A standard approach to generating decision trees is to learn them from a set of given examples. A disadvantage of this approach is that once a decision tree is learned, it is difficult to modify or to learn new decision trees that suit different decision-making situations. Such problems arise whenever there is incomplete or unusual reasoning situation. For example, when an attribute assigned to some node cannot be measured, or there is a significant change in the costs of measuring attributes or in the frequency distribution of events from different decision classes. Some attempts have been made to solve such problems using probabilistic or pruning techniques, but due to the inflexibility of the examples, it is difficult to learn a new decision tree for the given situation.

An attractive approach to resolving this problem is to learn and store knowledge in the form of decision rules, and to generate from them, whenever needed, a decision tree that is most suitable for a given situation. An additional advantage of such an approach is that it facilitates building *compact decision trees*, which can be much simpler than the logically equivalent conventional decision trees (by compact trees are meant decision trees that may contain branches assigned a *set of values*, and nodes assigned *derived* attributes, i.e., attributes that are logical or mathematical functions of the original ones). This section describes an efficient method, AQDT-1, that takes decision rules generated by an AQ-type learning system (AQ17), and builds from them a decision tree satisfying a given optimality criterion. The method can work in two modes: the *standard mode*, which produces conventional decision trees, and *compact mode*, which produces compact decision trees.

### *9.2 The AQDT-1 Algorithm*

AQDT-1 constructs a decision tree from decision rules by recursively selecting at each step the "best" attribute according to the attribute ranking measure described in (Imam, and Michalski, 1993), and assigning it to the new node. The process stops when the algorithm creates terminal branches that are assigned decision classes.

To facilitate such a process, the system creates a special data structure for each concept description (ruleset). This structure has fields such as the number of rules, the number of conditions in each rule, and the number of attributes in the rules. The system also creates an array of attribute descriptions. Each attribute description contains the attribute's name, domain, type, the number of legal values, a list of the values, the number of rules that contain that attribute, and values of that attribute for each rule. The attributes are arranged in the array in a lexicographic order, first, in descending order of the number of rules that contain that attribute, and second, in ascending order of the number of the attribute's legal values.

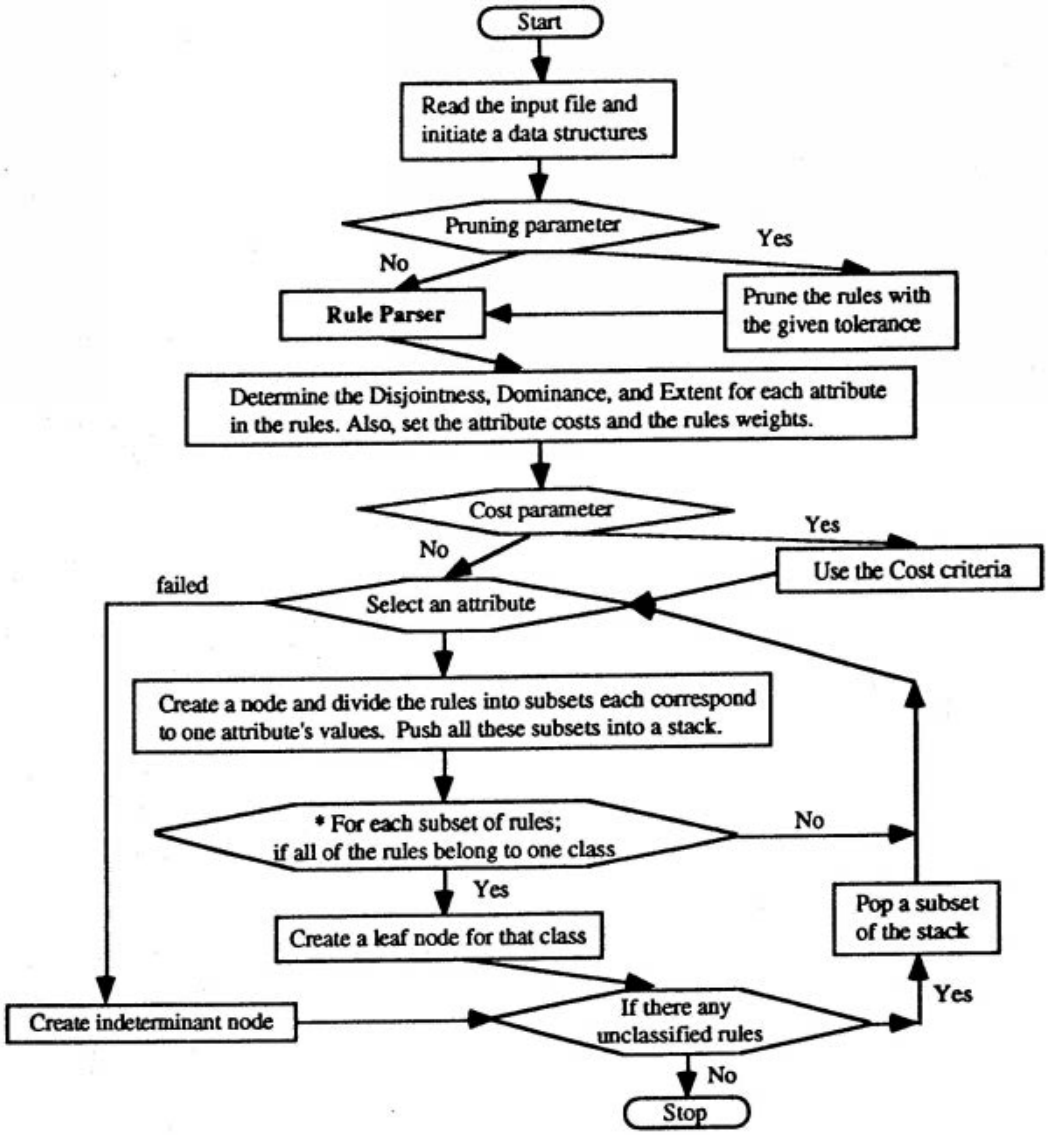



*Figure 2e.* The AQDT-1 Algorithm

---

The system can work in two modes. In the *standard* mode, the system generates standard decision trees, in which each branch has a specific attribute value assigned. In the *compact* mode, the system builds a decision tree that may contain "or" branches, i.e., branches assigned an internal disjunction of attribute values, whenever it leads to simpler trees.

To generate decision trees from rules, the method uses characteristic descriptions generated in the "dc" (disjoint cover) mode of AQ17. The reason for using characteristic descriptions is that they offer a greater choice of attributes in the process of building a decision tree, and this may lead to simpler decision trees. The reason for disjoint rulesets is that they are more suitable for building

decision trees, as the latter are equivalent to sets of logically disjoint descriptions. **Is this required?**

Assume that the input contains characteristic descriptions of the given decision classes. The description of each class is in the form of a ruleset. Assume that this set is the initial *ruleset context.*

*Step 1:* Evaluate each attribute occurring in the ruleset context using the LEF attribute ranking measure. Select the highest ranked attribute. Suppose it is attribute A.

*Step 2:* Create a node of the tree (initially, the root, afterwards, a node attached to a branch), and assign to it the attribute **A**. In the standard mode, create as many branches from the node as there are legal values of the attribute **A,** and assign these values to the branches. In the compact mode, create as many branches as there are disjoint value sets of this attribute in the decision rules, and assign these sets to the branches.

*Step 3:* For each branch, associate with it a group of rules from the ruleset context that contain a condition satisfied by the value(s) assigned to this branch. For example, if a branch is assigned values i of attribute **A**, then associate with it all rules containing condition [**A= i v** ...]. If a branch is assigned values i v j, then associate with it all rules containing condition [**A= i v j v** ...]. Remove from the rules these conditions. If there are rules in the ruleset context that do not contain attribute **A,** add these rules to all rule groups associated with the branches stemming from the node assigned attribute A. (This step is justified by the consensus law: [x=1] ≡ { [x=1] & [y= a] v [x=1] & [y=b] }, assuming that a and b are the only legal values of y.) All rules associated with the given branch constitute a ruleset context for this branch.

*Step 4:* If all the rules in a ruleset context for some branch belong to the same class, create a leaf node and assign to it that class. If all branches of the trees have leaf nodes, stop. Otherwise, repeat steps 1 to 4 for each branch that has no leaf.

## *9.3 DT Tables Guides*

### 9.3.1 dtarameters

The mandatory dtparameters table contains values which control the execution of the DT module. The set of DT parameters is given below.

**Compact (no)**

Specifies the format of the output decision tree. Given the value "no", DT generates a conventional decision tree. Given the value "yes", DT generates compact decision trees (see section 9.2).

**Cost (no)**

Specifies the LEF mode for selecting an attribute. Legal values are "yes" and "no". Given the value "no", cost information is not used when evaluating attributes. Given the value "yes" cost information is used.

**Prun (no)**

Specifies whether or not rule pruning shouldbe performed before generating a decision tree. Legal values are "yes" or "no"

**LEF_tol (0.0)**

Specifies the tolerance of the LEF. The possible values of this parameter are real numbers from 0.0 to 1.0.*** integer??**

**Prun_tol (1)**
Specifies the rule pruning performed. The value indicates the minimum number of examples that a rule must cover in order not to be removed. Prun must be "yes" for prun_tol to be used.

## 10. Testing Rules

### *10.1 Introduction*

ATEST is a program for evaluating the the performance of a rule base. ATEST has the ability to check both the predictive accuracy and the completeness and consistency of a rule base. ATEST takes as input rules, attribute definitions and control parameters. Each of these input types are discussed fully below. This description is taken from (Reinke, 1984).

### *10.2 Method*

In ATEST the fundamental operation is the calculation of the degree of match between a rule and a vector of attribute values (example). This value is called a degree of consonance. The calulation of consonance is dependent on parameter settings. Six parameters control rule evaluation. These six parameters are andtype, ortype, norm, threshold dropa2 and dweight. Three parameters control which features of ATEST are used in a specific run. These are test, misclass and cc.When rule testing is performed ATEST applies each rule to the pre-classified testing example. For each example a set of rules whose degree of consonance is within tau of the highest match is calculated. These rules are are said to be in the same 'rank'.

For a rule to be satisfied, some complex in the rule must be satified. A complex is satisfied if every selector in that complex is satisfied by the example under consideration. A selector may be considered to be a boolean conditional or a continuously-valued conditional.

### *10.3 ATEST tables*

ATEST takes as input parameters, domain descriptions and testing examples. The parameter descriptions are described below. The domain descriptions are identical to those described in section 4.4 to 4.7. The testing examples are provided in -test tables. These tables are identical in syntax to -tevents tables except that -test replaces -tevents. ATEST parameters should be in filestem.atparam. Domain descriptions should be in filestem.domain and testing data in filestem.test.

The parameters table is mandatory for ATEST. Default values (as shown in the parentheses) are used if no values are given. At least two parameters must be explcitly provided in the input file, however. The ATEST parameters are given below.

**test (yes)**
This parameter controls whether to test rules on the provided events. Legal values are: "yes", "no", and "sum". If "sum" is given a summary of the results for each class is reported. If "yes" is given a confusion matrix for each testing class (classes as columns and examples as rows) is reported. If "no" is given the rules are not evaluated against the examples.

**misclass (no)**
If an exanple is misclassified and misclass is on, then a trace of that event will be reported. If misclass is off, no such trace will be reported.

**cc (no)**
This parameter determines if completeness and consistency checking will be on.

**andtype (minimum)**
This parameter controls how conjunctions are evaluated. Legal values are "average" and "minimum".

**ortype (maximum)**
This parameter controls how disjunctions are evaluated. Legal values are "maximum and "psum" (probabilistic sum).

**norm (no)**
This parameter controls whether selectors of linear variables will be evaluated as boolean or continuously-valued (normalized) conditions. Legal values are "yes" (normlized) and "no" (boolean).

**threshold (0.50)**
This real-valued parameter controls the threshold for rule satisfaction. If rule consonance is greather than threshold, then the rule is satisfied by the event.

**dropa2 (1.00)**
This real-valued parameter controls when to stop using the $\alpha 2$ weight in rule evaluation. This value specifies the truth threshold a module must exceed before that module can be included in the cumulative weight of evidence. This is a measure of 'cumulative evidence'.

**dweight**
This real-valued parameters us used to determine which modules (rule-sets) are used for multiplying during completeness and consistency checking. The dweight is a measure of how important a particular complex is in reaching a decision.

## 11. Future Work

There are a number of new features in this implementation of AQ. Future work will involve better explaining when these features will be appropriate to use. The basic outline of this work was given in AQ17-MCI (Bloedorn, Michalski, Wnek, 1993). In this approach the system uses a meta-level description of the data to learn rules for the application of various representation space modifiers (RSM). This assistance will allow the novice user to better utilize the power of AQ.

## 12. Bibliography

## Appendix A. Programmer's Guide to Data Structures

### *1. Introduction*

The most important change in the implementation involved the binary representation of the complex/selector/event data. The original Pascal data structure supporting the complex/selector/event limited the size of individual structures, as well as the number of structures. This was due to the static allocation of memory. This was not a trivial limitation. On the PC version this limit was 200 events, and 12 variables. The new C version uses dynamic memory allocation. This allows the user to analyze much bigger input data.

As in the original Pascal implementation decision classes are organized hierarchically in a tree structure. Each node in the tree represents a class. A complex is a variant record representing a complex from a rule or an individual selector. As opposed to the original implementation, event information is stored in a separate structure. Future implementation will also prvide a separate structure for selector information. This change reduces memory use and improves speed.

The essential building block of the complex/selector/event structure is called bits. It is the address of the contiguous memory location holding the complex information. Consider simple event space as follows:

There are 4 variables: x1, x2, x3, x4.

Each of these variables has the following domain sizes:
x1-3 (0..2)
x2-2 (0..1)
x3-4 (0..3)
x4-2 (0..1)

The complex Comp has the following variable values:
x1=1, x2=0, x3=3, x4=1

The size of the complex is 3+2+4+2=11.

Two bytes (16 bits) will be allocated for storage. The last five last bits of the second byte are unused and are zeroed. The complex bytes will look as follows:

01010000 10100000

### *2. New Data Types*

typedef unsigned char * bits
This is the essential data type in the new implementation of AQ. Every variable of this type holds the pointer to the contiguous memory area storing the complex/selector/event.

```
typedef struct Complex {
pcomplex next;
...
union {
struct {
  bits compexbits;
   ...
```

```
}
struct {
 bits selectorbits;
  ...
}
}v;
}Complex;
typedef struct Complex *pcomplex;
```

Data type pcomplex is holds the pointer to the variant record containing information about complex and selector. In addition to storing the core logical part of complex or selector it holds some other data needed in the algorithm. The previous utilization of AQ also included event information in this data type . This version has separate data type holding the event information. In future implementation the pcomplex data type will pertain only to the complex from a rule. Selector data will be kept by separate structure.

```
typedef struct Event {
pevent next;
bits eventbits;
....
} Event;
typedef struct Event *pevent;
```

Type pevent holds the pointer to the data area containing information about events. Note that the eventbits part of Event is of the same type as complexbits and selectorbits in the Complex type.

```
typedef struct locate_ {
int firstbyte;
unsigned char firstbit_;
}
```

Type locate_ holds the pointer to the memory area containing information about variable configuration in the complex/selector/event. Firstbyte holds the byte number of the complex where the feature bits (variable data) starts . Firstbit holds the bit number where the feature starts.

### *3. New Global Variables*

The size of the complex/selector/event is determined in the setup function and is represented by the global variable size_of_complex_in_bits and size_of_complex_in_bytes. The bitwise location of each feature is kept by the dynamic array pointed by firstbit. The size of each variable is kept in the dynamic array pointed by varsize. In addition to these variables there is a structure type locate_ holding the bytewise and bitwise locations of each variable. This approach reduces the overhead associated with the access to features. Also there is no longer need to create special bitwise masks to access the features like in previous implementations. Some operations in the new implementation are done without entering the internal structure of the particular complex/selector/event - I will call such operations bytewise as opposed to bitwise functions that deal with the internal "partitioning" of the complex/selector/event .

From previous example:
comp = 01010000  10100000

If the presented complex name is comp and the memory location of its first byte is 100. The allocated space for it consists of two bytes.

comp = 100 (address of the first byte in complex)
size_of _complex_in_bits = 11
size_of_complex_in_bytes = 2
number_of _variables = 4

The domain size of each variable is represented by the pointer varsize. In this case :

*varsize = 3 (size of first domain)
*(varsize + 1) = 2 (size of second domain)
*(varsize + 2) = 4 (size of third domain)
*(varsize + 3) = 2 (size of fourth domain)

Global variable positions_ is the pointer to data type locate_.

positions_->firstbyte = 0 (the byte number where the first feature starts)
(positions_ +1)->firstbyte = 0 (the byte number where the second feature starts)
(positions_ +2)->firstbyte = 0 (the byte number where the third feature starts)
(positions_ +3)->firstbyte = 1 (the byte number where the fourth feature starts)

positions_->firstbit = 0 (bit location of the first feature)
(positions_ +1)->firstbit = 3 (bit location of the second feature)
(positions_ +2)->firstbit = 5 (bit location of the third feature)
(positions_ +3)->firstbit = 1 (bit location of the fourth feature)

Global variable firstbit is the pointer to the array containing the bitwise distances between the features and the beginning of complex/event/selector.

*firstbit = 0 (bitwise distance between first feature and the beginning of complex)
*(firstbit+1) = 3 (bitwise distance between second feature and the beginning of complex)
*(firstbit+2) = 5 (bitwise distance between third feature and the beginning of complex)
*(firstbit+3) = 9 (bitwise distance between fourth feature and the beginning of complex)

### *4. New Functions*

Most of our effort was put to streamline the basic set and logical operations in the new C version. Following you will find short descriptions of some new functions associated with the core of AQ. Most of the "high level" functions like traversetree, formrule, coverll etc remained unchanged.

The core of new logical and set operations is included in the new file called aqs.c. This file contains routines that deal with complexes at the lowest level. They can be viewed as a set of basic tools or drivers needed for the new C version of AQ.

void compl_allocation(bits *c)

This routine allocates the space needed to hold the complex/selector/event pointed by c. It uses the global variable size_of_complex_in_bytes.

void put_1(bits c, int _bytenr, unsigned char _bitnr)

This function places one in location pointed by the above parameters. C is the pointer to the beginning of the complex/selector/event, _bytenr represents the byte number in complex under consideration, _bitnr is the bit number in the byte. Both _bytenr and _bitnr start with zero.

void put_0(bits c, int _bytenr, unsigned char _bitnr)
Function similar to put_1 but places zero in the location- pointed by the above parameters. Parameters are the same as in put_1.

void invert_bit(bits c, int _bytenr, unsigned char _bitnr)
This function inverts the bit value in location pointed by the above parameters. The parameters are the same as in put_1.

char check_bit_status(bits c, int _bytenr, unsigned char _bitnr)
Check_bit_status returns one if the bit pointed by the above parameters is set to one. Otherwise it returns zero. See put_1 for the description of parameters.

void allocate_complexes(void)
This function is called by setup after the size of the complex is determined. It allocates complexes used locally in frequently called functions.

void create_bytes(void)
Create_bytes is called by setup. After domain and variable sizes are acquired create_bytes establishes the configuration of the complex/selector/event structure. This function creates global variables holding information of the bytewise and bitwise locations of each feature. (See GLOBAL VARIABLES)

void intersect_compl(bits c1, bits c2, bits c2)
The function intersects logically complexes c1 and c2 and places the result complex under c3.

int subset_compl(bits c1, bits c2)
Subset_compl returns one if complex c1 is a proper subset of complex c2, otherwise zero is returned.

char check_if_sel_fill(bits a, int fid)
The function returns one if complex a has all bits of feature number fid set to 1. Fid must be in range of zero to number_of _variables-1.

void copy_sel(bits a, bits b, int fid)
Copy_sel copies all bits assigned to feature fid of complex a to the same feature number of complex b. Fid must be in range of zero to number_of_variables-1.

void blank_feature(bits c, int fid)
Blank_feature sets all bits belonging to the feature fid to zero. Fid must be in range from zero to number_of_variables - 1

void copy_complex(bits c1, bits c2)
This function copies the complex pointed by c1 to c2. The operation is bytewise, this means that no feature check is performed. It is different from original Pascal implementation where such operation had to performed on the array of sets.

void intersect_compl(bits c1, bits c2, bits c3)
The function multiplies logically complexes c1 and c2 and puts the result into c3. Operation is bytewise.

## Appendix B. Sample Application

This appendix provides samples of input and ouput files for all seven modules of AQ17.

### *B.1 Learning Module*

This section provides sample input and output files for use with the learning module of AQ17. This example does not show how all of the AQ17 tables may be used. Only the files used by the learning module are described in this section. To run AQ17 with only the learning module type: aq17 m1.

#### B.1.1 Input to learning module

Input to the aq17 learning module requires a learning parameter (.lparam), domain description (.domain) and training data (.data). Training data may also be provided in the -events table format. If that format is used, data should be in the m1.train file. In this example training data is provided in an m1.data file, so the program automatically generates an m1.training file.

##### B.1.1.1 Contents of m1.lparam

```
parameters
maxstar trim echo wts
 10  mini  0  cpx
```

##### B.1.1.2 Contents of m1.domain

```
variables
  #  type  levels  cost  name
  1  lin     4    1.00  x1.x1
  2  lin     4    1.00  x2.x2
  3  lin     3    1.00  x3.x3
  4  lin     4    1.00  x4.x4
  5  lin     5    1.00  x5.x5
  6  lin     3    1.00  x6.x6
```

##### B.1.1.3 Contents of m1.data

Only part of this file is shown here. The entire file contains 87 lines.

```
x1 x2 x3 x4 x5 x6 class
 3  2  1  1  4  2  1
 2  1  2  1  3  1  1
 1  2  2  3  2  2  1
 2  3  1  3  4  2  1
 1  2  1  1  4  2  1
 2  1  1  2  3  1  1
 1  3  1  3  2  2  1
 3  2  1  2  4  2  1
 2  1  2  1  4  2  1
 2  1  2  3  4  1  1
 1  2  2  3  3  2  1
 3  1  1  2  2  2  1
 2  1  1  1  3  2  1
```

```
.....
3 3 1 3 4 2 2
2 2 1 1 2 2 2
2 3 2 2 1 1 2
2 2 1 3 3 2 2
3 3 1 3 2 1 2
2 2 1 3 4 2 2
3 3 2 3 3 2 2
1 1 1 1 3 2 2
.....
```

### B.1.2 Ouput from learning module

The learning module outputs the rules that it generated from the training examples to m4.rule. It also echoes to the .rule file some of the input tables. Which tables are echoed depends on the value of the echo parameter in the m4.lparam file. The input tables are given in section B.1.1 so only the new outhypo table will be shown here.

```
class1-outhypo
 #  cpx
 1  [x1=1] [x2=2..3] [x5=2..4]  (t:22, u:22)
 2  [x1=2..3] [x2=1] [x5=2..4]  (t:11, u:11)
 3  [x1=3] [x2=2] [x5=2..4]  (t:5, u:5)
 4  [x1=2] [x2=3] [x5=2..4]  (t:5, u:5)
```

```
class2-outhypo
 #  cpx
 1  [x5=1]  (t:16, u:11)
 2  [x1=3] [x2=3]  (t:15, u:11)
 3  [x1=2] [x2=2]  (t:10, u:10)
 4  [x1=1] [x2=1]  (t:7, u:6)
```

### *B.2 DCI Module*

This section provides sample input and output files for use with the DCI module. This module modifies the training data by constructing new attributes. These new attributes are added to the training data, the testing data (if present) and the domain (-names, domaintypes and variables tables). Input to the DCI module is the m1.dparam. m1.domain and m1.train (and m1.test if present). The results of the DCI module are normally sent directly to the learning module. In order to better show what DCI does, the modified m1.domain, and m1.train file will be shown as output.

#### B.2.1 DCI input

DCI input consists of the tables in m1.dparam, m1.domain and m1.train. See Appendix B.1.1 for the m1.domain file.

##### B.2.1.1 m1.dparam

```
parameters
echo operations
pvnex  test1

test1-operations
 cost  name      threshold
 10.0  relation  0.1
 10.0  plus      0.5
 10.0  multiply  0.8
```

##### B.2.1.2 m1.train

This file contains the -event tables form of the training examples. These tables can be entered directly, or generated from a .data file. As in B.1, the below table is incomplete, but enough of it is given to show the correct syntax. The "..." shows where the file was truncated.

```
class1-events
 x1 x2 x3 x4 x5 x6
 3  2  1  1  4  2
 2  1  2  1  3  1
 1  2  2  3  2  2
 2  3  1  3  4  2
 1  2  1  1  4  2
 2  1  1  2  3  1
 1  3  1  3  2  2
 3  2  1  2  4  2
 2  1  2  1  4  2
 2  1  2  3  4  1
...

class2-events
 x1 x2 x3 x4 x5 x6
 3  3  1  3  4  2
 2  2  1  1  2  2
 2  3  2  2  1  1
```

2 2 1 3 3 2
1 1 1 1 3 2
2 2 2 1 3 2
...

### B.2.2 DCI ouput

DCI modifies the training data, and attributes given to it. To show this, the modified m1.domain and m1.train files are given below.

#### B.2.2.1 m1.domain

variables

| # | type | levels | cost | name |
|---|---|---|---|---|
| 1 | lin | 4 | 1.00 | x1 |
| 2 | lin | 4 | 1.00 | x2 |
| 3 | lin | 3 | 1.00 | x3 |
| 4 | lin | 4 | 1.00 | x4 |
| 5 | lin | 5 | 1.00 | x5 |
| 6 | lin | 3 | 1.00 | x6 |
| 7 | lin | 3 | 3.00 | x1RELx2.x1RELx2 |
| 8 | lin | 3 | 3.00 | x1RELx5.x1RELx5 |
| 9 | lin | 3 | 3.00 | x3RELx5.x3RELx5 |
| 10 | lin | 3 | 3.00 | x5RELx6.x5RELx6 |

x1RELx2-names

| value | name |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 3 |

x1RELx5-names

| value | name |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 3 |

x3RELx5-names

| value | name |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 3 |

#### B.2.2.2 m1.train

class1-events

| x1 | x2 | x3 | x4 | x5 | x6 | x1RELx2 | x1RELx5 | x3RELx5 | x5RELx6 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 1 | 4 | 2 | 3 | 2 | 2 | 3 |
| 2 | 1 | 2 | 1 | 3 | 1 | 3 | 2 | 2 | 3 |
| 1 | 2 | 2 | 3 | 2 | 2 | 2 | 2 | 1 | 1 |
| 2 | 3 | 1 | 3 | 4 | 2 | 2 | 2 | 2 | 3 |
| 1 | 2 | 1 | 1 | 4 | 2 | 2 | 2 | 2 | 3 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 1 | 2 | 3 | 1 | 3 | 2 | 2 | 3 |
| 1 | 3 | 1 | 3 | 2 | 2 | 2 | 2 | 2 | 1 |
| 3 | 2 | 1 | 2 | 4 | 2 | 3 | 2 | 2 | 3 |
| 2 | 1 | 2 | 1 | 4 | 2 | 3 | 2 | 2 | 3 |

...

class2-events

| x1 | x2 | x3 | x4 | x5 | x6 | x1RELx2 | x1RELx5 | x3RELx5 | x5RELx6 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 3 | 1 | 3 | 4 | 2 | 1 | 2 | 2 | 3 |
| 2 | 2 | 1 | 1 | 2 | 2 | 1 | 1 | 2 | 1 |
| 2 | 3 | 2 | 2 | 1 | 1 | 2 | 3 | 3 | 1 |
| 2 | 2 | 1 | 3 | 3 | 2 | 1 | 2 | 2 | 3 |
| 3 | 3 | 1 | 3 | 2 | 1 | 1 | 3 | 2 | 3 |

...

### *B.3 HCI Module*

This section provides sample input and output files for use with the HCI module. This module modifies the training data by constructing new attributes and removing irrelevant attributes. These new attributes are added, or removed from the training data, the testing data (if present) and the domain (-names, domaintypes and variables tables). Input to the HCI module is the m1.hparam. m1.domain and m1.train (and m1.test if present). The results of the HCI module are normally sent directly to the learning module. In order to better show what HCI does, the modified m1.domain, and m1.train file will be shown as output.

#### B.3.1 HCI input

HCI input consists of the tables in m1.hparam, m1.domain and m1.train. See Appendix B.1.1 for the m1.domain file, and B.2.1 for the m1.train file.

##### B.3.1.1 m1.hparam

```
parameters
mode trim wts echo expand
ic  mini cpx pq   all
```

#### B.3.2 HCI ouput

HCI modifies the training data, and attributes given to it. To show this, the modified m1.domain and m1.train files are given below. A record of the sequence of attribute removals and additions is provided in m1.log

##### B.3.2.1 m1.domain

```
domaintypes
  type  levels  cost  name
  lin     4    1.00  x1
  lin     4    1.00  x2
  lin     3    1.00  x3
  lin     4    1.00  x4
  lin     5    1.00  x5
  lin     3    1.00  x6
  lin     2    1.00  class11
  lin     2    1.00  class22

 variables
  #  type  levels  cost  name
  1  lin     4    1.00  x1.x1
  2  lin     4    1.00  x2.x2
  3  lin     5    1.00  x5.x5
  4  lin     2    1.00  class11.class11
  5  lin     2    1.00  class22.class22
```

### B.3.2.2 m1.train

The below table is incomplete, but enough of it is given to show the correct syntax. The "..." shows where the file was truncated.

class1-events

| # | x1 | x2 | x5 | class11 | class22 |
|---|---|---|---|---|---|
| 1 | 3 | 2 | 4 | 0 | 0 |
| 2 | 2 | 1 | 3 | 1 | 0 |
| 3 | 1 | 2 | 2 | 1 | 0 |
| 4 | 2 | 3 | 4 | 1 | 0 |
| 5 | 1 | 2 | 4 | 1 | 0 |
| 6 | 2 | 1 | 3 | 1 | 0 |
| 7 | 1 | 3 | 2 | 1 | 0 |

...

class2-events

| # | x1 | x2 | x5 | class11 | class22 |
|---|---|---|---|---|---|
| 1 | 3 | 3 | 4 | 0 | 1 |
| 2 | 2 | 2 | 2 | 0 | 1 |
| 3 | 2 | 3 | 1 | 0 | 1 |
| 4 | 2 | 2 | 3 | 0 | 1 |
| 5 | 3 | 3 | 2 | 0 | 1 |
| 6 | 2 | 2 | 4 | 0 | 1 |
| 7 | 3 | 3 | 3 | 0 | 1 |
| 8 | 1 | 1 | 3 | 0 | 0 |
| 9 | 2 | 2 | 3 | 0 | 1 |

...

### B.3.2.3 m1.log

parameters

| run | mode | ambig | trim | wts | maxstar | echo | criteria | expand | window | th1 | th3 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | ic | neg | mini | cpx | 3 | pq | default | all | 1.00 | 0.67 | 0.10 |

class1-outhypo

| # | cpx |
|---|---|
| 1 | [x1=1] [x2=2..3] [x5=2..4] (total:22, unique:22) |
| 2 | [x1=2..3] [x2=1] [x5=2..4] (total:11, unique:11) |
| 3 | [x1=3] [x2=2] [x5=2..4] (total:5, unique:5) |
| 4 | [x1=2] [x2=3] [x5=2..4] (total:5, unique:5) |

class2-outhypo

| # | cpx |
|---|---|
| 1 | [x5=0..1] (total:16, unique:11) |
| 2 | [x1=3] [x2=3] (total:15, unique:11) |
| 3 | [x1=2] [x2=2] (total:10, unique:10) |
| 4 | [x1=0..1] [x2=0..1] (total:7, unique:6) |

Removed attributes: 3 4 6

class1-outhypo
 # cpx
 1 [x1=0..1] [x2=2..3] [x5=2..4] (total:22, unique:13)
 2 [x1=0..2] [x2=3] [x5=2..4] (total:14, unique:5)
 3 [x1=2..3] [x2=0..1] [x5=2..4] (total:11, unique:8)
 4 [x1=3] [x2=0..2] [x5=2..4] (total:8, unique:5)

class2-outhypo
 # cpx
 1 [x5=0..1] (total:16, unique:11)
 2 [x1=3] [x2=3] (total:15, unique:11)
 3 [x1=2] [x2=2] (total:10, unique:10)
 4 [x1=0..1] [x2=0..1] (total:7, unique:6)

Removed attributes:

Relevant attributes: 1 2 5

class1-outhypo
 # cpx
 1 [x1=0..1] [x2=2..3] [x5=2..4] (total:22, unique:13)
 2 [x1=0..2] [x2=3] [x5=2..4] (total:14, unique:5)
 3 [x1=2..3] [x2=0..1] [x5=2..4] (total:11, unique:8)
 4 [x1=3] [x2=0..2] [x5=2..4] (total:8, unique:5)

class2-outhypo
 # cpx
 1 [x5=0..1] (total:16, unique:11)
 2 [x1=3] [x2=3] (total:15, unique:11)
 3 [x1=2] [x2=2] (total:10, unique:10)
 4 [x1=0..1] [x2=0..1] (total:7, unique:6)

class11-attribute
 # cpx
 1 [x1=0..1] [x2=2..3] [x5=2..4]
 2 [x1=2..3] [x2=0..1] [x5=2..4]
 3 [x1=0..2] [x2=3] [x5=2..4]

class22-attribute
 # cpx
 1 [x5=0..1]
 2 [x1=3] [x2=3]
 3 [x1=2] [x2=2]

Relevant attributes: 1 2 5 7 8

```
class1-outhypo
 #  cpx
 1  [class11=1]  (total:38, unique:11)
 2  [x2=2..3] [class22=0]  (total:32, unique:5)


class2-outhypo
 #  cpx
 1  [class22=1]  (total:37, unique:36)
 2  [x1=0..1] [x2=0..1]  (total:7, unique:6)

This run used (milliseconds of CPU time):
 System time:       0
 User time :      350
```

### *B.4 NT Module*

This section provides sample input and output files for use with the NT module. This module modifies the training data by removing training examples which are covered by 'light' rules in the learned covers. Input to the NT module is the ex1.domain and ex1.train (and ex1.test if present) files. The results of the NT module are normally sent directly to the learning module. In order to better show what NT does, the modified ex1.train file will be shown as output.

#### B.4.1 NT input

NT input consists of the tables in ex1.domain and ex1.train. In this example nt is invoked with th=10. This threshold is entered at the command line:

```
> aq17 -n 10 ex1

AQ17 [release 1] Multistrategy Learning System
------------------------------------------------------------------------

     Options:
          noise removal

running the NT module...
 1 example removed from training  data

learning rules. Please standby...

 c_1-outhypo
  #  cpx
  1  [height=4..5]  (t:2, u:2)
  2  [length=4]  (t:2, u:2)


 c_2-outhypo
  #  cpx
  1  [length=5..6] [width=2]  (t:2, u:2)
  2  [length=1..2]  (t:2, u:2)

Done learning rules

aq17 Learning System Exiting
```

##### B.4.1.1 ex1.domain

```
variables
  #  type  levels  cost  name
  1  lin     20    1.00  height
  2  lin     20    1.00  length
  3  lin     20    1.00  width
```

### B.4.1.2 ex1.train

```
c_1-events
 height length width
5 10 2
4 5 3
5 8 4
3 4 1
1 4 2


c_2-events
 height length width
3 6 2
2 5 2
3 2 1
1 1 0
```

### B.4.2 NT ouput

NT modifies the training data. To show this, the modified ex1.train file is given below. The first example of class c_1 has been removed.

```
c_1-events
 height length width
4 5 3
5 8 4
3 4 1
1 4 2


c_2-events
 height length width
3 6 2
2 5 2
3 2 1
1 1 0
```

## *B.5 SCALE Module*

This section provides sample input and output files for use with the SCALE module. This module modifies the training data by discretizing the attribute values. The new discretized attribute values replace the values given in the training data, and the testing data. Input to the SCALE module is the t2.sparam and t2.train (and t2.test if present) files. The results of the SCALE module are normally sent directly to the learning module. In order to better show what SCALE does, the modified t2.train file will be shown as output with the t2.intv file which show into which intervals the values were rewritten.

### B.5.1 SCALE input

SCALE input consists of the tables in t2.sparam and t2.train.

#### B.5.1.1 t2.sparam

```
intervals       1
8
name        min_intv   max_intv   method   chi_thresh
x1          2          50         chi      1.0
x2          2          50         chi      10.0
x3          2          50         chi      10.0
x4          2          50         chi      10.0
x5          2          50         chi      10.0
x6          2          50         chi      10.0
x7          2          50         chi      25.0
x8          2          50         chi      1.0
```

#### B.5.1.2 t2.train

```
class_1-events
 x1   x2   x3    x4    x5   x6   x7    x8
 165.1 581.2 1572.4 2471.0 280.1 264.9 1081.5 812.3
 164.9 540.4 1671.2 2036.2 298.5 292.1 1025.1 807.6
 196.8 733.8 1416.4 1626.2 246.1 251.5 953.9  937.5
 165.6 572.8 1908.1 2212.2 289.7 275.2 1037.2 947.8
 163.0 510.9 2139.0 2207.8 319.8 355.0 1011.5 1231.3
 150.3 459.0 2052.3 2297.1 303.0 282.0 1354.4 1042.0
 166.0 575.9 2121.8 1809.7 256.8 267.3 983.9  1007.7
 190.6 675.7 1619.8 2064.0 254.9 251.3 1040.9 934.4
 164.7 560.0 1575.9 1568.3 267.1 250.5 858.7  793.6
 142.3 481.9 1367.6 1296.7 237.3 208.1 765.0  830.4
 149.9 499.4 1535.8 1553.7 193.0 226.7 840.5  903.0
 194.3 698.6 2347.8 1751.7 249.2 307.3 780.3  1068.3
 187.3 671.0 1476.3 1880.8 260.2 235.4 1011.4 844.3
 183.0 660.6 1346.3 1526.2 302.4 302.6 931.2  936.6


class_2-events
 x1   x2   x3    x4    x5   x6   x7    x8
 111.8 341.2 1610.1 1721.4 197.6 215.6 746.0  859.7
 146.1 464.4 2878.6 1766.7 194.5 203.5 693.3  1205.2
```

135.4 427.6 1956.0 1500.7 187.7 225.8 698.4 1095.5
109.1 372.9 1686.1 1245.5 148.4 176.8 502.6 676.7

### B.5.2 SCALE ouput

SCALE modifies the attribute values in the training (and possible testing) data. To show this, the modified t2.train file and t2.intv file is given below.

#### B.5.2.1 t2.train

class1-events

| x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 |
|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 2 | 2 | 2 | 2 | 2 |
| 4 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 4 | 2 | 1 | 1 | 2 | 2 | 2 | 4 |
| 4 | 2 | 2 | 2 | 2 | 2 | 2 | 4 |
| 4 | 2 | 2 | 2 | 2 | 2 | 2 | 6 |
| 4 | 1 | 2 | 2 | 2 | 2 | 2 | 4 |
| 4 | 2 | 2 | 2 | 2 | 2 | 2 | 4 |
| 4 | 2 | 2 | 2 | 2 | 2 | 2 | 4 |
| 4 | 2 | 1 | 1 | 2 | 2 | 2 | 2 |
| 2 | 2 | 1 | 1 | 2 | 1 | 2 | 2 |
| 4 | 2 | 1 | 1 | 1 | 2 | 2 | 4 |
| 4 | 2 | 2 | 1 | 2 | 2 | 2 | 4 |
| 4 | 2 | 1 | 2 | 2 | 2 | 2 | 2 |
| 4 | 2 | 1 | 1 | 2 | 2 | 2 | 4 |

class2-events

| x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 1 | 1 | 1 | 1 | 3 |
| 3 | 1 | 2 | 1 | 1 | 1 | 1 | 5 |
| 1 | 1 | 2 | 1 | 1 | 1 | 1 | 5 |
| 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 |

### B.5.2.2 t2.intv

SCALE discretizes the initial attribute values into the intervals defined in the following ranges. This file is produced both for the user to see into what intervals the original data was discretized, and to allow testing data to be scaled to the same intervals.

```
attribute intervals-x1
  1   0.00... 142.29
  2 142.30... 146.09
  3 146.10... 149.89
  4 149.90... 196.80

attribute intervals-x2
  1 166.00... 481.89
  2 481.90... 733.80

attribute intervals-x3
  1   0.00... 1610.09
  2 1610.10... 2878.60

attribute intervals-x4
  1 1245.50... 1809.69
  2 1809.70... 2471.00

attribute intervals-x5
  1 148.40... 237.29
  2 237.30... 2207.80

attribute intervals-x6
  1 176.80... 226.69
  2 226.70... 1520.72

attribute intervals-x7
  1   0.00... 764.99
  2 765.00... 1354.40

attribute intervals-x8
  1 676.70... 793.59
  2 793.60... 859.69
  3 859.70... 902.99
  4 903.00... 1095.49
  5 1095.50... 1231.29
  6 1231.30... 1231.30
```

## *B.6 ATEST Module*

This section provides sample input and output files for use with the ATEST module. This module tests rules against labelled testing examples. Input to the ATEST module is the ex1.tparam ex1.domain and ex1.test files. The results of the ATEST module is a confusion matrix which records the degree of match between each testing example and each class. Summary statistics on the percent correct, percent first-rank correct (flexible match) and percent strict-match correct are given, for this example, in the ex1.test-results file.

### B.6.1 ATEST input

ATEST input consists of the tables in ex1.atparam, ex1.domain, ex1.test and ex1.rule.

#### B.6.1.1 ex1.atparam

parameters
run test echo threshold tau
1 yes p 0.5 0.00

#### B.6.1.2 ex1.domain

domaintypes

| type | levels | cost | name |
|---|---|---|---|
| lin | 20 | 1.00 | height |
| lin | 20 | 1.00 | length |
| lin | 20 | 1.00 | width |

variables

| # | type | levels | cost | name |
|---|---|---|---|---|
| 1 | lin | 20 | 1.00 | height.height |
| 2 | lin | 20 | 1.00 | length.length |
| 3 | lin | 20 | 1.00 | width.width |

#### B.6.1.3 ex1.test

c_1-test

| # | height | length | width |
|---|---|---|---|
| 1 | 3 | 10 | 2 |
| 2 | 5 | 2 | 3 |
| 3 | 3 | 5 | 4 |
| 4 | 6 | 1 | 1 |

c_2-test

| # | height | length | width |
|---|---|---|---|
| 1 | 5 | 2 | 2 |
| 2 | 3 | 5 | 3 |

### B.6.1.4 ex1.rule

```
c_1-outhypo
 #  cpx
 1  [length=8,10]
 2  [height=0]
 3  [height=9]


c_2-outhypo
 #  cpx
 1  [height=2..3,5] [length=2,5..6]
```

## B.6.2 ATEST ouput

The generated summary statistics and confusion matrix is put in the file ex1.test-results.

### B.6.2.1 ex1.test-results

```
parameters
test  misclass tau   andtype ortype  threshold norm cc  dweight dropa2
yes   false    0.00  average maximum  0.50      no  no  0.50    1.00


DECISION CLASSES
----------------
D1 = c_1
D2 = c_2

*****************************************
                    TEST RESULTS FOR CLASS c_1

*****************************************


   CORRECT DECISION CLASS = D1(c_1)
I=================================================I
I EVENT     : #TIES :  ASSIGNED DECISION          I
I           :       :  D1   D2                    I
I=================================================I
I c_1-1     :       : *1.00* 0.50                 I
I-------------------------------------------------I
I c_1-2     :       : 0.00  1.00                  I
I-------------------------------------------------I
I c_1-3     :       : 0.00  1.00                  I
I-------------------------------------------------I
I c_1-4     :  1    : 0.00  0.00                  I
I=================================================I
I     # 1st RANK EVENTS:   1    2                I
I=================================================I
TOTAL # 1st RANK EVENTS/#EVENTS =  3/ 4 = 0.75
NUMBER OF EVENTS SATISFYING CORRECT RULE :  1
```

```
*****************************************
                TEST RESULTS FOR CLASS c_2
*****************************************

  CORRECT DECISION CLASS = D2(c_2)
I=================================================I
I EVENT     : #TIES :  ASSIGNED DECISION          I
I           :       :  D1   D2                    I
I=================================================I
I c_2-1     :       : 0.00 *1.00*                 I
I-------------------------------------------------I
I c_2-2     :       : 0.00 *1.00*                 I
I=================================================I
I    # 1st RANK EVENTS:  0    2                   I
I=================================================I
TOTAL # 1st RANK EVENTS/#EVENTS =  2/ 2 = 1.00
NUMBER OF EVENTS SATISFYING CORRECT RULE :  2


*****************************************
    TEST RESULTS -- SUMMARY
*****************************************

OVERALL % CORRECT : 50.00
OVERALL % CORRECT 1ST RANK : 50.00
OVERALL % CORRECT ONLY CHOICE : 50.00

Number of testing events satisfying individual complexes
in the correct class description:

                    COMPLEXES
                     C 1  C 2  C 3
CLASS   c_1             1    0    0
CLASS   c_2             2

Note: the entries (class x, CI) in the above table show the number
    of testing events of class x that were completely covered by
    the ith complex (CI)

This run used (milliseconds of CPU time):
CPU user time:      16 milliseconds
CPU system time:     0 milliseconds
```

### *B.7 DT-Module*

This section provides sample input and output files for use with the DT module. This module constructs decision trees from decision rules. Input to the DT module is the m1.dtparam, m4.domain and m1.rule. The results of the DT module are given in m1.tree.

#### B.7.1 DT input

DT input consists of the tables in m1.dtparam, m1.domain and m1.rule.

##### B.7.1.1 m1.dtparam

```
parameters
run  mode  cost  rul_prun  cost_tol  disj_tol  prun_tol
 1    dc    no    no          no        no        no
```

##### B.7.1.2 m1.domain

```
variables
  #  type  levels  cost  name
  1  nom     4     1.00  x1
  2  nom     4     1.00  x2
  3  nom     3     1.00  x3
  4  nom     4     1.00  x4
  5  nom     5     1.00  x5
  6  nom     3     1.00  x6
```

##### B.7.1.3 m1.rule

```
Pos-outhypo
  #  cpx
  1  [x1=1v2v3][x2=1v2v3][x3=1v2][x4=1v2v3][x5=1][x6=1v2]
     (Total:xx, Unique:yy)
  2  [x1=1][x2=1][x3=1v2][x4=1v2v3][x5=1v2v3v4][x6=1v2]
     (Total:xx, Unique:yy)
  3  [x1=2][x2=2][x3=1v2][x4=1v2v3][x5=1v2v3v4][x6=1v2]
     (Total:xx, Unique:yy)
  4  [x1=3][x2=3][x3=1v2][x4=1v2v3][x5=1v2v3v4][x6=1v2]
     (Total:xx, Unique:yy)


Neg-outhypo
  #  cpx
  1  [x1=1][x2=2v3][x3=1v2][x4=1v2v3][x5=2v3v4][x6=1v2]
     (Total:xx, Unique:yy)
  2  [x1=2][x2=1v3][x3=1v2][x4=1v2v3][x5=2v3v4][x6=1v2]
     (Total:xx, Unique:yy)
  3  [x1=3][x2=1v2][x3=1v2][x4=1v2v3][x5=2v3v4][x6=1v2]
     (Total:xx, Unique:yy)
```

### B.7.2 DT ouput

DT generates a decision tree from rules. The contents of m4.tree are given below.

```
x5=1 :   Neg
x5 = 2:
|       x1 = 1:
|       |       x2 = 1 :   Neg
|       |       x2 = 2 :   Pos
|       |       x2 = 3 :   Pos
|       x1 = 2:
|       |       x2 = 1 :   Pos
|       |       x2 = 2 :   Neg
|       |       x2 = 3 :   Pos
|       x1 = 3:
|       |       x2 = 1 :   Neg
|       |       x2 = 2 :   Pos
|       |       x2 = 3 :   Neg
x5 = 3:
|       x1 = 1:
|       |       x2 = 1 :   Neg
|       |       x2 = 2 :   Pos
|       |       x2 = 3 :   Pos
|       x1 = 2:
|       |       x2 = 1 :   Pos
|       |       x2 = 2 :   Neg
|       |       x2 = 3 :   Pos
|       x1 = 3:
|       |       x2 = 1 :   Neg
|       |       x2 = 2 :   Pos
|       |       x2 = 3 :   Neg
x5 = 4:
|       x1 = 1:
|       |       x2 = 1 :   Neg
|       |       x2 = 2 :   Pos
|       |       x2 = 3 :   Pos
|       x1 = 2:
|       |       x2 = 1 :   Pos
|       |       x2 = 2 :   Neg
|       |       x2 = 3 :   Pos
|       x1 = 3:
|       |       x2 = 1 :   Neg
|       |       x2 = 2 :   Pos
|       |       x2 = 3 :   Neg
```