The AQ19 System for Machine Learning and Pattern Discovery:

A General Description and User's Guide

Ryszard S. Michalski
Kenneth A. Kaufman

MLI 01-2

THE AQ19 SYSTEM FOR MACHINE LEARNING AND PATTERN DISCOVERY:

A GENERAL DESCRIPTION AND USER'S GUIDE

Ryszard S. Michalski* and Kenneth A. Kaufman

Machine Learning and Inference Laboratory
George Mason University
Fairfax, VA 22030-4444

MLI 01-2
P01-2

March 2001

* Also Institute of Computer Science, Polish Academy of Sciences

The AQ19 System for Machine Learning and Pattern Discovery:

A General Description and User's Guide

Summary

This report provides a description and a user's guide for AQ19, a program for machine learning and pattern discovery. AQ19 works in two modes: *Theory Formation* and *Pattern Discovery*. In Theory Formation mode, given examples of two or more concepts, AQ19 hypothesizes general descriptions of these concepts optimized according to a modifiable criterion of hypothesis preference. In Pattern Discovery mode, given data with indicated input and output variables, AQ19 determines strong patterns in the relationship between the input and output variables.

AQ19 aims at performing *natural induction*, that is, a form of inductive inference that seeks hypotheses that appear natural to people, and thus are easy to comprehend and interpret. This feature is achieved in AQ19 by employing a highly expressive description language, called *attributional calculus*. Descriptions or patterns generated by AQ19 are in the form of *attributional rulesets* that have a higher expressive power than decision trees and decision rules used in standard rule learning programs. In addition to the central module for machine learning and pattern discovery, AQ19 also incorporates auxiliary modules for ruleset testing (ATEST) and for attribute selection (PROMISE).

AQ19 can work with different kinds of data, from very small to very large, containing noise or noise-free, definite or ambiguous, and involving different types of attributes. In addition to nominal, discrete, and continuous attributes, it can also reason with *structured attributes*, whose value sets are hierarchies. AQ19 enables a user to experiment with a variety of criteria for description optimality, generate rulesets at different levels of generality, and of different types (unordered or ordered, discriminant or characteristic, or optimized according to a user-defined multi-criterion description quality measure).

In contrast to conventional learning systems that assume that the output (dependent) variable represents a fixed number of disjoint classes (more than one), AQ19 can also be applied to problems with only one class (no negative examples), with overlapping classes (which occur frequently, e.g., in design problems), or with hierarchically arranged classes. AQ19 is written in ANSI-C, so it may be compiled and run on a variety of platforms.

Keywords: Machine learning, data mining, inductive inference, learning from examples.

Acknowledgments

The authors express their gratitude to all the researchers who were involved in the development and implementation of various components of AQ19, in particular, Eric Bloedorn, Mark Maloof, Janusz Wnek, and Qi Zhang. They also thank numerous students who implemented earlier versions of the AQ learning methodology.

AQ19 is a member of the AQ family of symbolic learning systems, whose beginnings go to the early 70s. Throughout the years, many generations of students and researchers in the U.S. and abroad have been implementing or re-implementing various versions of the AQ learning methodology, which pioneered the progressive *covering* (a.k.a. "separate and conquer") approach to machine learning.

Research on the development of the AQ methodology and AQ programs has been conducted in the Machine Learning and Inference Laboratory at George Mason University and previously at the Artificial Intelligence Laboratory at the University of Illinois at Champaign-Urbana. Recent research was supported in part by the National Science Foundation under Grants No. IIS-9906858 and IIS-9904078, in part by the Office of Naval Research under Grant No. N00014-91-J-1351, in part by the Defense Advanced Research Projects Agency under Grant No. N00014-91-J-1854, administered by the Office of Naval Research, and Grant No. F49620-92-J-0549 administered by the Air Force Office of Scientific Research, and in part by the grant LUCITE Task #32 with the University of Maryland, Baltimore County.

Table of Contents

1 INTRODUCTION 1

2 SUMMARY OF FEATURES 1

2.1 NATURAL INDUCTION 1
2.2 FLEXIBLE CRITERION OF RULE OPTIMALITY 3
2.3 SPECIFICATION OF THE INTER- AND INTRA-RELATIONSHIPS IN RULESETS 4
2.4 DISCRIMINANT VS. CHARACTERISTIC DESCRIPTIONS: CONTROLLING THE LEVEL OF GENERALITY 5
2.5 RULESET EVALUATION AND TESTING (ATEST CLASSIFIER MODULE) 5
2.6 HANDLING OF CONTINUOUS ATTRIBUTE DOMAINS 6
2.7 OVERALL SUMMARY 6

3 WHAT IS NEEDED TO RUN AQ19 7

3.1 OVERVIEW OF REQUIREMENTS 7

4 GETTING STARTED: A SIMPLE EXAMPLE OF USING AQ19 7

4.1 INPUT TO THE LEARNING MODULE 8
4.2 OUTPUT FROM THE LEARNING MODULE 8
4.3 AN EXAMPLE USING THE TESTING MODULE 9
4.3.1 Input to AQ19 with testing events 9
4.3.2 Output from AQ19 with testing results 11
4.4 A BRIEF REVIEW OF RULESET REPRESENTATION 14

5 FILE FORMATS AND EXPLANATIONS 15

5.1 THE PARAMETERS TABLE 16
5.2 THE CRITERIA TABLES 23
5.3 THE DOMAINTYPES TABLE 24
5.4 THE VARIABLES TABLE 25
5.5 THE NAMES TABLES 27
5.6 THE STRUCTURE TABLES 27
5.7 THE INHYPO AND OUTHYPO TABLES 29
5.8 THE EVENTS AND TEVENTS TABLES 30
5.9 THE CHILDREN TABLES 31
5.10 THE VARSEL TABLE 32

6 TESTING RULESETS 32

6.1 OBJECTIVE 32
6.2 TESTING METHODS 33

7 AQ19'S PREDECESSORS: A LITTLE HISTORY FOR THE CURIOUS 35

REFERENCES 37

1 INTRODUCTION

The tremendous growth of machine learning applications, in particular in the areas of data, image and text mining, has created an urgent need for systems that can efficiently derive general hypotheses or strong patterns from collections of data. In many such applications, hypothesized descriptions need not only have high predictive accuracy, but also be easy to interpret and comprehend by the user. In addition, different applications may require different kinds of descriptions, depending on the problem at hand, the measurement cost of attributes, the frequency of decision classes, the level of noise, missing values, the method of applying descriptions in decision making, and other factors.

After a general description or a pattern has been generated, an obvious next step is to test its predictive accuracy. Therefore, a learning system should have a link to a module for testing the learned descriptions. The AQ19 program represents an effort to address the issues.

To help the reader quickly understand the scope of AQ19 capabilities, the next section reviews its main features. A theoretical foundation and algorithms underlying these features are provided in (Michalski, 2001). The following sections describe various aspects of the program and illustrate its performance on selected problems.

2 SUMMARY OF FEATURES

2.1 Natural induction

The primary emphasis of current methods for symbolic learning has been on the task of efficiently inducing descriptions or patterns with high predictive accuracy. As mentioned above, many applications require, however, not only accurate but also comprehensible descriptions, easily interpretable by the user. The latter requirement, although frequently brought up in the literature, has been largely ignored in practice. This is evidenced by the fact that authors of papers reporting results on inductive learning or data mining rarely show the actual decision trees or rules learned by the system, and usually limit themselves to listing only comparative statistics about their predictive accuracy.

A likely reason for the above is that results of learning are not infrequently large decision trees or large sets of conventional[*] decision rules that are difficult to understand, and difficult to turn into a cognitive model. In many cases, such results are a direct consequence of relatively low expressive power of knowledge representations used by the learning system. The "low representational power" means here that expressing some cognitively simple descriptions may require a highly complex representational structure (a large decision tree, or many conventional decision rules). Consequently, such complex structures are difficult to understand although they may represent cognitively simple concepts.

To illustrate the above idea, consider a decision rule:

$$\text{If } x_1 \leq x_2,\ x_3 \neq x_4, \text{ and } x_3 \text{ is red or blue, then decision is A} \qquad (1)$$

[*] By conventional decision rules are meant rules CONDITION => DECISION, in which CONDITION is a conjunction of *atomic conditions* in the from "attribute-value" or "attribute-rel-value", where rel is $\geq$ or $\leq$.

It is easy to see that if variables x_i, i=1,2,3,4, are two-valued, representing (1) would then require a decision tree with 26 nodes and 20 leaves, or 12 conventional decision rules. If variables x_i were five-valued, then representing (1) would require a decision tree with 810 leaves and 190 nodes, or 600 conventional decision rules. The need for such a complex representation of a relatively simple relationship demonstrates a significant limitation of these representational formalisms.

If a learning system could express the above rule in a form similar to (1), that is, in a "natural" form for people, then resulting representation would be easier to understand and interpret by a user. A learning system that aims at satisfying such a requirement (in addition to the requirement of high predictive accuracy) is called a *natural induction* system (Michalski, 2001).

The central design criterion for the AQ19 system is to employ a knowledge representation language that would facilitate natural induction. Such a language must use structures and operators that approximate corresponding natural language concepts, be syntactically and semantically well-defined, and relatively easy to implement.

To approach this goal, AQ19 employs *the attributional calculus*, a highly expressive description language based on variable-valued logic system VL1 (Michalski, 2001). Attributional calculus can be viewed as a logic system that conceptually occupies a place between propositional calculus and predicate logic. For example, attributional calculus expression of (1) would be:

$$[\text{Decision} = \text{A}] \ <= \ [x_1 \leq x_2] \ \& \ [x_3 \neq x_4] \ \& \ [x_3 = \text{red v blue}] \qquad (2)$$

Rule (2) reads: Decision is A if x1 ≤ x2, x3 ≠ x4, and x3 is red or blue. The symbol "v" that links different attribute values is called an internal disjunction, in order to distinguish it from the conventional ("external") disjunction, which links truth-valued expressions in propositional and predicate calculus. The external disjunction is also a part of the attributional calculus. It is worth noting that natural language employs both internal and external disjunction.

In order to provide a user with more information about rule (1), AQ19's parameters can be set up so that the rule is printed in a form resembling:

[Decision = A] if $[x_1 \leq x_2$: 1998, 866] & $[x_3 \neq x_4$: 800, 419] & $[x_3$= red or blue: 780, 420]
(t: 750, u:700, n:14, f:4, q=.92) (3)

-- bracketed statements indicate elementary statements (conditions) in the rule,
-- a pair of numbers after ":" in every condition specifies the number of positive and negative examples covered by the condition, respectively
-- *t* specifies the total number of positive examples covered by the rule (also known in various research communities as the *rule coverage* or *support*)
-- *u* specifies *unique coverage*: the number of examples covered by this rule, and by no other rule associated with Decision=A
-- *n* denotes the number of negative examples covered by the rule (also known as *negative coverage*, *exceptions*, or *conflicts*)
-- *f* denotes the number of examples in the training set that are matched *flexibly* (Section 6)
-- *q* denotes the *rule quality* as measured by a rule quality criterion based on rule coverage and *training accuracy* (Kaufman and Michalski, 1999)

In addition, there is an external rule representation, which is more informative and easier to interpret for the user, in which rule (3) would be presented in the form:

==========

R1. Decision is A if:		Supp	NegCov	Conf
1. $x_1 \leq x_2$		1998	866	70%
2. $x_3 \neq x_4$		800	419	66%
3. x_3 is red or blue		780	420	65%
	Rule	750	14	

with the following annotations:

--*Evaluations:* flexibly matched =4, Rule quality=.92
--*Ambiguous:* a list of covered training examples that belong to both positive and negative classes
--*Exceptions:* (false positives) is a list of pointers to the negative examples covered by the rule
--*Positives covered:* (true positives) is a list of pointers to positive examples crisply covered by the rule.
--*Flexibly covered:* a list of pointers to positive examples that are covered by flexible matching (Section 6.2). (4)

==========

The form (4) is based on the rule representation employed in the INLEN system for multistrategy data mining and decision-making (Kaufman, 1997; Michalski and Kaufman, 1998)

To distinguish between constituent conditions in conventional decision rules (which are in the form *attribute rel value*, where *rel* is =, >, or <), from more elaborate conditions in attributional rules, the former are called atomic and the latter are called composite (Michalski, 2001). As illustrated by the example above attributional rules with composite conditions may be able to represent some functions much more compactly and understandably than conventional decision rules. Attributional rules reduce to conventional decision rules when each constituent condition in them is atomic.

In AQ19, a concept description is in the form of a set of attributional rules, which is equivalent to a disjunctive normal form in attributional calculus. Output from AQ19 is in the form of a family of rulesets, where each ruleset is a collection of attributional rules whose union characterizes one decision class.

2.2 Flexible criterion of rule optimality

Symbolic machine learning programs typically employ only one criterion that controls their function. In decision tree learning, it is an attribute selection criterion, and rule learning it is some measure of rule quality. For example, C4.5 uses an information gain ratio as a criterion for attribute selection (Quinlan, 1993). The rule learner, RIPPER-k, evaluates rules according to a measure that determines the difference between the numbers of positive and negative examples covered by a rule (Cohen, 1995); and the rule learner, CN2, uses an entropy-based measure (Clark and Niblett, 1989).

It is unlikely that any such single criterion will be suitable for all possible applications. Consequently, the performance of these programs depends on the problem at hand. In contrast to such a single-criterion philosophy, AQ19 makes available to a user a menu of eleven different criteria for evaluating rule quality. A user can select the most appropriate criteria from the list, and arrange them into a lexicographic evaluation functional (LEF), which best represents the requirements of the problem (Section 5.2). To make it easy to use by an inexperienced user, the system has heuristic rules that automatically specify LEF, when the user does not specify it explicitly.

2.3 Specification of the inter- and intra-relationships in rulesets

In different application areas, different types of descriptions may be most suitable for the given problem. Yet, conventional rule or decision tree learners create descriptions of one type only, regardless of the problem. Consequently, such results may be good for one class of problems, but less adequate for another. In short, these methods do not provide a user with the possibility of fitting the type of the description to the problem.

Let us explain the above point in more detail. In applications in which each example can belong to one class only (a common classification problem), a desirable form of a description would be a logically disjoint and complete family of rulesets, that is, a family in which no two rulesets logically intersect, and the union of all the rulesets covers the whole representation space, or at least the subset from which all events can reasonably occur. In such a case, each possible example is uniquely classified to one class. A decision tree constitutes such a classifier. An assumption is made here that any possible example belongs to one of the a priori defined classes.

Under this assumption, however, one can implement two solutions. One solution is that descriptions of concepts do not intersect over examples from different classes, but may intersect logically over spaces not containing any training examples (this is the Intersecting Covers or ic mode). The non-empty intersection of two descriptions, say, of concepts A and B, make the descriptions more conservative. If a new, unseen example falls into the area of their intersection, then the system will answer A or B (as both descriptions will be matched), rather than risk an incorrect decision. Another solution under the single class assumption is to ensure that class descriptions are logically disjoint (this is the Disjoint Covers or dc mode). Learning in ic mode has the advantage that the rules learned will be independent of the order in which the classes are learned, while in dc mode, that may not be the case; rules will be bound by the constraint that they must not intersect with previously learned rules for other classes.

The ic mode may be also used in situations where the output attribute is set-valued. This means that a given example or event may legitimately belong to more than one class. This situation occurs, for example, in such applications as engineering design, in which a given function may be performed by more than one tool (e.g., Moczulski, 1998). To be able to generate such alternative decisions, the ic mode is used. In contrast to the previous case, here the intersections of rulesets cover cases that belong to corresponding classes (rather then areas unoccupied by training examples). Thus, if there are 10 classes, and any non-empty subset of them can potentially be assigned to an example, one may need only 10 rulesets for representing all possible decision subsets (while methods that are limited to producing logically disjoint classes, e.g., decision trees, would require separate learning of all relevant subsets of classes, that is, up to 1023 classes).

In some applications, it is desirable to learn an unordered family of rulesets. Given such a family, and an example to classify, the rulesets can be matched against the example in any order. This may provide a useful flexibility (at the expense of representational complexity). If, on the other hand, classes are semantically ordered (the dependent variable is of rank type), or if it is useful to consider them in a certain order (e.g., in the order of decreasing class frequencies), it may be desirable to generate an ordered family of rulesets (this is done by using the sequential cover, or sc, mode in AQ19 (a.k.a. decision list mode—see the description of the mode parameter in Section 5.1). An ordered family of rulesets can be significantly simpler than a logically equivalent unordered family of rulesets (in terms of the total number of rules and/or conditions in them), but its evaluation requires a strict order in which rules are tested against data.

2.4 Discriminant vs. characteristic descriptions: controlling the level of generality

Another important aspect of descriptions to be learned is their intended purpose and level of generality. When a description is used only to discriminate between a given class and other classes from an a priori defined set, one seeks a *discriminant* description, which contains the minimum number of rules and conditions that are sufficient for discrimination (the *discriminant cover*; Michalski, 2001). In some applications, one may want to find a description that contains the maximum number of features that are common in a class of examples, that is, to determine a *characteristic description* of the class (the *characteristic cover*). Disregarding special cases, discriminant descriptions are the most general descriptions of given classes of examples, and characteristic descriptions are the most specific descriptions of these classes. Characteristic descriptions can be particularly useful in applications in which the number of classes is not fixed a priori (as typically assumed in current learning methods), but is growing (e.g., in document classification, or computer intrusion detection). Sometimes the most desirable is a description of intermediate generality. AQ19 allows one to control the type of description (the *genlevel* parameter; see Section 5.1).

2.5 Ruleset evaluation and testing (ATEST classifier module)

The ATEST module is used to apply a knowledge base to a set of examples. If the classification of the examples is known, this serves as a test of the predictive accuracy of the knowledge. If the classification is not known, ATEST serves as classifier of unknown examples. The ATEST module makes five different testing methods available (Section 6).

- Strict vs. flexible matching

In some applications, classification decisions should made by evaluating descriptions (rulesets) as logical statements, which either match or do not match a given example (*crisp or strict matching*). In other applications, given an unclassified example, a classification decision is made by evaluating the degree of match between the example and candidate classes, and determining the best match (flexible matching). The crisp matching method is computationally much simpler than the flexible matching method, especially if there are very many classes. Also, cognitively, it is easier to understand the decision process when the crisp method is used. On the other hand, the flexible matching method tends to be more accurate than the crisp method in complex problems (as it is able to draw more complex boundaries between concepts in the representation space).

2.6 Handling of Continuous Attribute Domains

A major difference between AQ19 and its predecessor AQ18 is in the programs' handling of continuous numeric attributes. In AQ18 and previous versions, these attributes needed to be discretized (made discrete), either using intervals specified by the user, or through a data-driven statistical evaluation (Kerber, 1992). The data structures of AQ19 allow the extend-against operator (Michalski, 1983) to be applied directly to the real-valued domain in order to generate maximal intervals (or sets of intervals) that in conjunction with other conditions in the rule do not cover the negative examples. These maximal intervals are reduced in the final rule according to the *epsilon* parameter, which specifies whether the interval should be left as is, trimmed back to minimal size such that all covered examples remain covered, or trimmed back some proportion of that distance.

2.7 Overall Summary

For a detailed explanation of the theory, algorithms and different features underlying the AQ approach to natural induction, and their embodiment in AQ19, the reader is referred to (Michalski, 2001). An experimental comparison of the rule quality criterion employed in AQ19 with criteria used in other learning methods is described in (Kaufman and Michalski, 1999).

AQ19 learns task-optimized attributional rulesets from examples and counter-examples, and/or from previously known rules (learned or handcrafted). Each ruleset is a collection of rules (a ruleset) for one decision class. When learning rulesets, AQ19 employs background knowledge in the form of 1) definitions of attributes and their types, 2) prior rules (input hypotheses) when available, and 3) a user-defined preference criterion that ranks competing candidate hypotheses.

The preference criterion is composed by arranging available elementary preference criteria into a lexicographic evaluation functional, called LEF (Section 5.2). For the simplicity of using AQ19, a default preference criterion is applied, unless the user creates a different one, aiming at reflecting better the given learning task.

AQ19 has a number of features that were not present in previous versions of AQ programs. To give an overall understanding of AQ19's capabilities, below is a summary of its major functions and features with a brief explanation.

- Selection of a subset of attributes based on user criteria in problems with a very large number of attributes.
- Modes of both crisp and flexible matching for rule application or testing.
- Capability for an automatic ruleset simplification (by removing rules), without introducing any misclassification of training examples (the *no-loss truncation* due to flexible matching)
- Full functionality for reasoning with hierarchically structured decision attributes
- Ability to take into consideration different types of input attributes in inductive reasoning: nominal, rank, cyclic, structured and continuous.
- Ability to apply a data-driven approach to the handling of continuous attributes.
- Rule testing confusion matrices can be output at three levels of abstraction.
- Training and testing examples can be weighted based on frequency of occurrence.
- A "uniclass" mode for characterizing a single set of examples.

- Five different options for dealing with ambiguous examples, i.e., examples that occur in more than one class.
- Ability to learn rulesets that are complete and consistent with regard to all the training data.
- Ability to handle noise and inconsistency in the data, which is achieved by applying a description quality criterion that maximizes the rule "strength" at the expense of rule completeness and/or inconsistency.
- Generating rulesets that optimize a user-designed preference criterion, assembled from the set of available elementary rule evaluation criteria, in order to maximally fit the specific learning problem.
- Ability to generate rulesets with different relationships among individual rules: intersecting, disjoint, or ordered.
- Ability to generate rulesets at different level of generality: maximally general, maximally specific, intermediate, and optimized according to a given rule quality criterion.
- To make the system easy for use by inexperienced users, it can be run without defining any parameters (in this case, the system dynamically determines which parameters are most suitable for a given dataset, or uses default parameters).

The primary output from the program is a family of rulesets. Each ruleset is a collection of attributional rules (VL1* based rules) associated with one decision class (that is, with one value of the decision variable). Thus, there are as many rulesets as decision classes.

Depending on the control parameters (Section 5), AQ19 can produce outputs with more or less information about the learned rulesets. The underlying algorithm for generating rulesets is AQ, which is the first known "separate and conquer" (a.k.a. progressive covering) method for rule induction. It performs a stepwise beam search through a space of attributional rules, until it finds a ruleset that satisfies the termination criterion (Michalski, 2001).

3 WHAT IS NEEDED TO RUN AQ19

3.1 Overview of Requirements

AQ19 can be run on SUN-Solaris 1 and 2), PC-Windows -and Macintosh operating systems. To run it, do as follows:

1. Get/install/access a copy of the system, which consists of one or more executable files.
2. Build the input files required for your task.
3. Enter the appropriate command.

4 GETTING STARTED: A SIMPLE EXAMPLE OF USING AQ19

This section describes a sample use of AQ19 using the m1 files provided in the AQ19 package. m1 (or Monk #1) is the first in the set of three classification problems known collectively as the Monk's problems (Thrun et al., 1991). The goal of this problem is to correctly predict whether the given robot (as described in a single example) is in class1 or class2. Each robot is described by six attributes (x1..x6). This example does not show how all of the AQ19 tables are used, but it serves as an introductory example featuring some of the most important table types.

If the package has been successfully installed (see the installation notes if this is not the case) the example can be run as follows:

(*Solaris, Win*)	aq19 < m1.aqin	(with aq19 in the current directory or path)
(*Mac*)	double click on AQ19	(m1 must be in the same folder as AQ19)

The following sections describe the input file (m1.aqin) and output generated by AQ in more detail.

4.1 Input to the Learning Module

Input to AQ19 in this example includes a parameters table, variables table, and two events tables. Not all of the training events from the m1 file are shown here for space reasons. The output of AQ19 is shown in Section 4.2. The first table defines the parameters that will guide AQ19's performance (Section 5.1). The second table defines the six variable domains (Section 5.4). The remaining tables define the event sets (Section 5.8).

```
parameters
run maxstar  genlevel  echo  wts
1    10      mini      pve   cpx

variables
    #   type   size  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

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
 1  2  2  3  3  2
 3  1  1  2  2  2

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

4.2 Output from the Learning Module

This section shows what rules are generated after running AQ19 over the entire m1.aqin input file. Which tables are echoed depends on the value of the echo parameter in parameters table.

The echo parameter in the parameters table in the input file tells AQ19 to echo the parameters, variables and events tables in addition to the learned rules (outhypo tables). The input tables are already given in Section 4.1 so only the new outhypo table are shown here. The complete output is available in m1.aqout.

```
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)
```

Outhypo tables containing decision rules (covers) are generated for each class. The class1-outhypo table gives rules describing the class1 examples (robots). Similarly the class2-outhypo table provides rules for class2. The weights associated with rules describe the coverage of individual rules. For instance, the weights associated with the first rule for class2 indicates that 16 examples are satisfied by that rule, and 11 of these examples are satisfied by this rule only. More details on these weights are provided in the description of the 'wts' parameter in Section 5.1.

4.3 An example using the Testing Module

This section provides sample input and output files for use with the testing module of AQ19. This module tests rules against labeled testing examples. Input to the testing module consists of parameters, domains, rules or training events and testing events. The parameters are given to the system in a parameters table. The available parameters and their meanings are given in Section 5.1). The domains of variables are given in the standard variables table (Section 5.4). If rules have already been generated in a previous run or are from some other source, then AQ19 is run in test only mode (as indicated in the parameters table). If rules have not yet been generated, training events in an events table must then be provided. In this case rules will be generated and immediately evaluated against the testing data.

The amount of information supplied by the testing module, and the method of evaluation between testing examples and rules is controlled by the test parameter (Section 5.1). This section provides examples of each of the available testing options.

4.3.1 Input to AQ19 with testing events

This section provides an example of an AQ19 input, which contains testing examples. These files are provided under the name m1test.aqin and m1test.aqout. The evaluation methods designated by test parameter codes "e", "f", "m", "q" and "w" are invoked in sequence, and the "c" parameter is present telling the program to echo a full confusion matrix. When multiple test evaluation methods are selected, they are performed in the order that they appear in the parameters table. This example introduces two types of input tables not shown in Section 4.1: *names* tables, which define names for the different values of a variable (Section 5.5), and *tevents* tables, which provide the program with event sets by which rules are to be tested (Section 5.8).

```
parameters
  run   mode   ambig   genlevel   wts   maxstar  echo     criteria  verbose
     test
   1    ic     pos     spec   cpx     10     peq      default     1     efmqwc

  variables
    #   type     size   cost   name
    1   nom       3     1.00   hs
    2   nom       3     1.00   bs
    3   nom       2     1.00   sm
    4   nom       3     1.00   ho
    5   nom       4     1.00   jc
    6   nom       2     1.00   ti

 ti-names
value name
0     no
1     yes

 sm-names
value name
0     no
1     yes

hs-names
value name
0     round
1     square
2     octagonal

 bs-names
value name
0     round
1     square
2     octagonal

 ho-names
value name
0     sword
1     balloon
2     flag

 jc-names
value name
0     red
1     yellow
2     green
3     blue

Pos-events
    #   hs         bs         sm   ho       jc      ti
    1   round      round      yes  flag     yellow  yes
    2   round      round      yes  sword    red     no
    3   round      square     yes  sword    red     yes
    4   round      octagonal  yes  balloon  red     yes
    5   square     square     yes  balloon  red     yes
    6   square     square     no   balloon  green   yes

Neg-events
    #   hs         bs         sm   ho       jc      ti
    1   round      octagonal  yes  sword    yellow  no
    2   square     octagonal  yes  sword    yellow  no
    3   octagonal  square     no   sword    green   no
    4   octagonal  round      yes  sword    blue    yes
    5   octagonal  octagonal  no   balloon  green   no
```

```
    6   octagonal  round      no   balloon  blue    no
    7   octagonal  square     yes  flag     red     no
    8   octagonal  round      no   flag     green   no
    9   round      octagonal  no   flag     blue    yes
   10   round      octagonal  no   flag     green   yes
   11   square     round      yes  flag     yellow  yes

 Pos-tevents
    #   hs         bs         sm   ho       jc      ti
    1   round      octagonal  yes  sword    blue    yes
    2   round      octagonal  yes  sword    green   yes
    3   round      round      yes  sword    red     no
    4   round      square     yes  sword    red     yes
    5   square     square     no   balloon  green   yes

Neg-tevents
    #   hs         bs         sm   ho       jc      ti
    1   round      octagonal  yes  sword    yellow  no
    2   square     octagonal  yes  sword    yellow  no
    3   round      round      yes  sword    red     no
    4   octagonal  square     no   sword    green   no
    5   octagonal  round      yes  sword    blue    yes
```

4.3.2 Output from AQ19 with testing results

The output when testing events are included consists of the rulesets and the requested testing results. The testing results are displayed in a confusion matrix that provides for every testing event the degree of confidence that that event belongs to each class. If the event matches a class other than the class to which it belongs, the name of that best matched class will be printed under the 'Class' heading. If the event most matches the correct class, no entry is given in the Class column. If a testing example matches multiple classes including the correct class, that example will be counted as correctly classified. Below are listed the same set of testing examples tested using all five evaluation methods. This list includes flexible matching (test parameter setting "e"), weighted flexible matching (test parameter setting "f"), rform (test parameter setting "m"), hgform (test parameter setting "q") and igform (test parameter setting "w") (Sections 5.1 and 6). The c option was also given to the test parameter so that full confusion matrices would be provided for each testing summary. If there are many testing events, the user may prefer not to see the entire confusion matrix, but rather a summary of the tests. The file containing this output is found in m1test.aqout.

```
Pos-outhypo
  #  cpx
  1  [hs=round] [bs=round,square] [sm=yes] [ho=sword,flag] [jc=red,yellow]
     (t:3, u:3)
  2  [hs=round,square] [bs=square,octagonal] [ho=balloon] [jc=red,green]
     [ti=yes]  (t:3, u:3)

Neg-outhypo
  #  cpx
  1  [hs=round,octagonal] [jc=green,blue]   (t:7, u:7)
  2  [hs=square,octagonal] [sm=yes] [ho=sword,flag] [jc=red,yellow]
     (t:3, u:3)
  3  [hs=round] [bs=octagonal] [sm=yes] [ho=sword] [jc=yellow] [ti=no]
     (t:1, u:1)


*************************************************************
                    Testing Summary (FlexMatch)
*************************************************************
 # Event   Class          member   Pos      Neg
```

Pos:				
1	Neg		0.60(1)	1.00(1)
2	Neg		0.80(2)	1.00(1)
3			1.00(1)	0.75(2)
4			1.00(1)	0.75(2)
5			1.00(2)	0.50(1)
Neg:				
1			0.80(1)	1.00(3)
2			0.60(1)	1.00(2)
3	Pos		1.00(1)	0.75(2)
4			0.40(1)	1.00(1)
5			0.60(1)	1.00(1)

Events Correct: 7 # Events Incorrect: 3 Accuracy: 70%

Total Selectors: 22 # Total Complexes: 5

Testing Summary (Weighted FlexMatch)

# Event	Class	member	Pos	Neg
Pos:				
1	Neg		0.57(2)	1.00(1)
2	Neg		0.77(2)	1.00(1)
3			1.00(1)	0.68(2)
4			1.00(1)	0.68(2)
5			1.00(2)	0.56(1)
Neg:				
1			0.81(1)	1.00(3)
2			0.56(1)	1.00(2)
3	Pos		1.00(1)	0.68(2)
4			0.34(1)	1.00(1)
5			0.52(1)	1.00(1)

Events Correct: 7 # Events Incorrect: 3 Accuracy: 70%

Total Selectors: 22 # Total Complexes: 5

Testing Summary (Rform)

# Event	Class	member	Pos	Neg
Pos:				
1	Neg		0.00	0.64
2	Neg		0.00	0.64
3			0.50	0.00
4			0.50	0.00
5			0.50	0.00
Neg:				
1			0.00	0.09
2			0.00	0.27
3	Pos		0.50	0.00
4			0.00	0.64
5			0.00	0.64

```
 # Events Correct:    7  # Events Incorrect:   3   Accuracy:  70%
.....................................................................
 # Total Selectors:  22  # Total Complexes:    5
*********************************************************************


*********************************************************************
                    Testing Summary (Hgform)
*********************************************************************
 # Event   Class          member   Pos      Neg
---------------------------------------------------------------------
  Pos:
     1     Neg                     0.00     1.00
     2     Neg                     0.00     1.00
     3                             1.00     0.00
     4                             1.00     0.00
     5                             1.00     0.00
  Neg:
     1                             0.00     1.00
     2                             0.00     1.00
     3     Pos                     1.00     0.00
     4                             0.00     1.00
     5                             0.00     1.00
---------------------------------------------------------------------
 # Events Correct:    7  # Events Incorrect:   3   Accuracy:  70%
.....................................................................
 # Total Selectors:  22  # Total Complexes:    5
*********************************************************************


*********************************************************************
                    Testing Summary (Igform)
*********************************************************************
 # Event   Class          member   Pos      Neg
---------------------------------------------------------------------
  Pos:
     1     Neg                     0.00     0.41
     2     Neg                     0.00     0.41
     3                             0.18     0.00
     4                             0.18     0.00
     5                             0.18     0.00
  Neg:
     1                             0.00     0.06
     2                             0.00     0.18
     3     Pos                     0.18     0.00
     4                             0.00     0.41
     5                             0.00     0.41
---------------------------------------------------------------------
 # Events Correct:    7  # Events Incorrect:   3   Accuracy:  70%
.....................................................................
 # Total Selectors:  22  # Total Complexes:    5
*********************************************************************

This testing used:
  System time:      0.017 seconds
  User time:        0.00  seconds
```

In each of the test runs shown above, 7 of the 10 test events were correctly classified. Their degrees of match varied according to the testing algorithm.

4.4 A Brief Review of Ruleset Representation

AQ19 uses the VL_1 (Variable-valued Logic system 1) and APC (Annotated Predicate Calculus) (Michalski, 1975; 1983; 2001) representational formalisms.

Training examples are given to AQ19 in the form of *events*, which assign values to the domain's variables. Each *decision class* (or class, for short) in the training set may be assigned a set of events, which form the set of *positive examples* of the class. During the learning of rules, the events 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 AQ19 chooses the best decision rule set (according to user-defined criteria) that classifies the input events. 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 *candidate hypotheses* or *partial covers*.

Each decision rule is described by one or more conditions (also known as selectors), all of which must be met for the rule to apply. A *condition* is a relational statement and is defined as:

[*REFEREE REL REFERENCE*]

where:

REFEREE (LHS of a condition) is an attribute, or two or more attributes connected by "&" or "v" (called internal conjunction or disjunction, respectively)
REL (RELATION) is one of the following symbols: <, <=, =, <>, >=, >
REFERENCE (RHS of a condition) is a value, a range of values, an internal disjunction of values, or an attribute.

Simple conditions state that the attribute in REFEREE takes one of the values defined in REFERENCE. Examples of simple conditions (a.k.a. simple selectors) are shown below:

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

Extended conditions may be used when more than one attribute take on the same values. Attribute names are separated by a ".". Examples of extended conditions are shown below:

[hair_color.eye_color = brown] ("hair color and eye color are both brown")
[length.width.height = 10..15] ("the length, width, and height are all in the range 10 to 15")
[state_of_birth.state_of_residence = Georgia] (the state of birth and state of residence is Georgia)

A *rule* (also called a complex) is a conjunction of conditions. The following are examples of rules in AQ notation:

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

For simplicity, the conjunction symbol "&" is usually omitted.

A *ruleset (or a cover)* for a decision class is a set of rules that together describe all of the positive examples and none of the negative ones of that decision class. The following is an example of a ruleset or a cover:

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

A ruleset or cover is satisfied if any of its rules are satisfied. A rule is satisfied if all of its conditions are satisfied. A condition is satisfied if the referee takes one of the values in the reference. The ruleset shown in the above example can be interpreted as follows:

An object is a flag if:

1) Its color is 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.

5 FILE FORMATS AND EXPLANATIONS

The AQ learning and testing programs accept input of data in the formats described below.

AQ format input files consist of a set of tables, all of which must be separated by at least one blank line. Tables consist of three parts: their identifier (one line), their header (the second line), and their body. The identifier indicates the type of table, and if it is a type that can have multiple individual tables, (such as the names tables, for which there may be one for each attribute), it also includes the name specifying when this particular table applies. The name and the type are separated by a hyphen, e.g., "shape-names".

The header line identifies each of the columns in the body of the table; this is essential because columns may often be put in different orders, or even be omitted entirely. If an optional column is not used, a default value will be assigned. Names within a table header may be separated by spaces or tabs.

The body simply consists of the entries in the table, separated by spaces. Lines may be continued over a carriage return by ending the line with a backslash (\) character.

Comments can be placed anywhere in the input file. The syntax of comments is the same as the syntax for comments in C: /* comment */. Comments may span multiple lines. Comments will not be echoed back at output.

Example

```
/* Filename: m1.aqin
    Date: 1/23/98
    Source: Monk's problem set */
```

The following sections describe in detail the tables in AQ, their purpose, and syntax. Most tables are optional and many of the parts within each table are also optional.

5.1 The parameters table

The optional parameters table contains values that control the execution of AQ19. All of the parameters have default values. The default values are provided in parentheses following the name of the parameter. Each row of the parameters table corresponds to one run of the program. In this way the user may specify in a single input file many runs using different parameter settings on the same data.

An input file may include multiple parameters tables in various parts of the input file. However, there are a number of constraints and complexities if the user intends to use this feature directly.

- If the first parameter table has no 'run' field, i.e., no run numbers are specified, then no subsequent parameters table may include this parameter. Similarly, if the first parameter table includes a 'run' parameter, then all following tables must include it explicitly as well.
- Run numbers are used as labels for parameter settings. Run values must begin with 1 and each *new* run number must be one more than the highest one previously used. A previously used run number may appear on another parameter line, meaning that both parameter lines with this run number refer to the same run. If for that reason there are multiple values for any parameter in a given numbered run, the values appearing later will overwrite the earlier ones; the last value given supersedes the prior ones; this value will take effect. More details on the run parameter are given below.

A parameters table consists of a name line (that simply reads "parameters"), a header line defining the table's columns, and one or more lines defining parameter settings. The columns that may appear in a parameters table are as follows (with default values in parentheses):

ambig (neg)
An optional parameter that controls the way ambiguous examples (i.e. overlapping examples from both the positive and negative 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 set when consistent rules are learned.

pos — Ambiguous examples are always taken as positive examples for the current class, and are therefore covered by more than one classification rule set when complete rulesets are learned.

empty — Ambiguous examples are ignored, i.e., treated as though they were not part of the input event set. They may or may not be covered by some classification rule(s).

max — This option can be used when training examples are weighted according to number of occurrences. If there is one class for which there are more occurrences of the ambiguous example than in any other single class, learning will take place with the point in the event space covered by the ambiguous examples considered to belong to that class. Otherwise, the point will be handled as if the ambig parameter was set to empty.

maj — This option can be used when training examples are weighted according to number of occurrences. If there is one class for which there are more occurrences of the ambiguous example than in all the other classes combined, learning will take place with the point in the event space covered by the ambiguous examples considered to belong to that class. Otherwise, the point will be handled as if the ambig parameter was set to empty.

criteria (default)
Entry is the name of the criteria table (Section 5.2) to be applied to a given run. The name must be of alpha type, and a criteria table with that name (unless it is "default") must appear later in the input file.

echo (pvne)
Specifies which tables are to be printed as part of the 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 white space separates words in the input and will confuse the parser). Legal values for the echo parameter and the tables they represent are shown below:

0 ----- no echo
b ----- *children* tables
c ----- *criteria* table
d ----- *domaintypes* table
e ----- *events* tables
i ------ *inhypo* tables
n ----- *names* tables
p ----- *parameters* table
q ----- *tevents* tables
s ----- *structure* tables
v ----- *variables* table

genlevel(mini)
trim (mini)
Either of these two names specifies an optional parameter that specifies the generality of the output rules (i.e., the number of possible events they satisfy). The legal values are:

gen — Rules are as general as possible, involving the minimum number of conditions, each with a maximum number of values.

mini — Rules are as simple as possible, involving the minimum number of conditions, each with a minimum of values.

spec — Rules are as specific as possible, involving the maximum number of conditions, each with a minimum of values.

Each condition restricts the set of events that will satisfy a rule, but each value in a condition relaxes these restrictions. Hence, rules with few conditions, all of which permit many values will be more general than ones with many conditions, each of which specifies only a few values.

maxstar (1)
Optional parameter that specifies the maximum number of alternative solutions retained during each stage of rule generation. The program uses *a beam search*, in which at any intermediate stage, the best candidate hypotheses are retained up to a certain number. A higher number specifies a wider beam search, which also requires more computer resources and processing time. Empirical evidence indicates that in general the size of maxstar should be approximately the same as the number of variables used. The rules produced tend to indicate a good compromise between computational resources and rule quality. Maxstar values may range from 1 to 50. The default is the minimum value.

mode (ic)
An optional parameter that controls the way in which AQ19 is to form rules. Legal values for this column are:

ic — "Intersecting covers" mode allows rules from different classes to intersect over areas of the learning space in which there are no examples.

dc — "Disjoint covers" mode produces covers that do not intersect at all with one another.

vl — "Variable-valued logic" or sequential cover mode produces rules are order-dependent. That is, the rules for class *n* will assume that the rules for classes 1 to *n*-1 were not satisfied. Hence there are no rules given for the last class; if none of the other rules were satisfied, an example is by default put into this class. *Note: This option is not available for decision tree generation.*

To illustrate the difference between these modes, consider two classes, one consisting of red circles, and the other one consisting of blue squares. In ic mode, the rules "Class 1 if red, Class 2 if square" might be produced. In dc mode, such a ruleset would not be allowed, since red squares would be described by both rules. In vl mode, only the rule for Class 1 would be necessary; anything else would be assumed to belong to Class 2.

neg_ex_probe (0)
An optional parameter only employed when the *noise* parameter (see below) is set to yes, this parameter limits the application of the rule generation operator. In normal mode (no noise or when *neg_ex_probe* = 0), AQ19 finds rules that distinguish positive examples of the target class from each negative example in turn, choosing the best from among them. It may be the case that a near-optimal rule is learned after a relatively small number of negative examples are examined, and that the rest of the process is somewhat a waste of time. This parameter, when set to a positive value, terminates the process if ever the number of iterations specified by the parameter fail to generate any improvement in the rules under consideration.

noise (no)
An optional parameter that controls whether the rule creation process is to treat the training data as potentially containing noise or other variance (value yes), or as an infallible oracle of the data (value no). In the latter case, rules will always be perfectly consistent with regard to the training data (with allowances for ambiguity -- see the *ambig* parameter). When noise may be present, AQ19 may search for "better" rules that nonetheless admit some negative examples of the target class. Two methods are used to generate better rules that are inconsistent with regard to the input data. During rule generation, negative examples of the target class whose exclusion from potential rules does not lead to higher rule quality are treated as noise and ignored. Second, when a rule has been learned, AQ19 tries to generalize it further, even if that may introduce inconsistencies, if that will result in a better rule.

q_weight (0.5)
An optional parameter that affects the system's criterion for judging the quality of candidate rules when the **noise** parameter has been activated (set to *yes*). This weight defines the balance between a rule's coverage level and its training accuracy, with legal values between 0 (inclusive) and 1 (exclusive). The value indicates the portion of the rule's quality measure that will be based on its coverage level. For example, q_weight = 0.25 indicates that coverage will contribute 25% to the rule's quality measure and training accuracy the other 75%.

rule_probe (0 or 2)
rtol (0 or 20)
An optional pair of parameters used when a complete ruleset for each class (one in which every training example of the class satisfies at least one of the learned rules) is not desired. For simple rulesets, a complete ruleset is preferable, but in large, noisy data-mining applications, the generation of a complete ruleset will be a time-consuming process typically resulting in a few strong rules, and many spurious ones.

The use of these two parameters limits the search process. After each rule is generated, it is tested to determine whether its quality level is close enough to that of the best rule discovered so far (the necessary degree of closeness is defined by the *rtol* parameter) to warrant a continued search. After *rule_probe* consecutive failures, the rule generation process for the current decision class terminates. *Rtol* is a tolerance expressed in percent; a value of 30 indicates that any rule whose quality is within 30% of that of the best rule found thus far resets the termination clock.

In default mode (both set to zero), a complete ruleset is generated. If only *rule_probe* is set, *rtol* is assigned a default value of 20%. If only *rtol* is set, *rule_probe* is assigned a default value of 2.

run (1..n)
An optional parameter that controls for which execution of the program the parameters line applies. Run numbers must be positive integers beginning with "1", with no succeeding integer appearing as a run number before its predecessor has already appeared. If used, this parameter must be in the first column of the parameters table. The default value is simply the number of the line, so the third parameters line would be assigned a default run number of 3.

test (m)
Optional parameter that controls the method of evaluation and the form of the output produced from testing the provided tevents (testing events) against the learned or provided rules. If no tevents are given, this parameter has no effect. This parameter is specified by a string of characters from the set {a, b, c, e, f, m, q, w}. The characters e, f, m, q and w specify different methods for testing; the test runs will be performed in the order that they appear, allowing for a direct comparison between testing methods. The presence or absence of the a, b, or c character specifies the completeness of the output. The five methods vary in their evaluation of the degree of match between the testing examples and the rules. The legal values for this parameter and an explanation of each method of evaluation are as follows:

e	Equal-weight flexible matching. Degrees of match between rules and test events are determined by the percentage of conditions in the rule matched by the testing event.
f	Weighted flexible matching. Similar to mode e, except the weight of a given condition in determining a degree of match is based on its informativeness in explaining the training data set.
m	INLEN, or rform mode used for testing. This is the mode used in the INLEN system (Michalski et al., 1992).
q	hgform evaluation method first described in (Michalski and Chilausky, 1980).
w	igform evaluation method described in (Michalski et al, 1986).

The last three methods shown above use a two-step approach to determining the degree of match between the testing examples and the different decision classes. First the examples are checked for *exact match* (i.e., satisfying the rule completely). If any exact match is found, their degree of match with the various decisions are solely based on the rules that were matched exactly. If there was no exact match, *flexible matching* is called upon (using the same algorithm as mode e), in which the degree of match is instead based on how closely the rules come to covering the testing examples.

As noted above, the presence or absence of the values a, b, or c controls the degree of detail in the output:

a, b, c	These values control whether a confusion matrix is reported in addition to the summary of the test run. If the value c is present, the full matrix is echoed, consisting for each test event of the event identifier, the class to which it was assigned (if not the one it was intended for), whether or not it was exactly matched by a rule (not applicable in modes e and f), and the example's degree of match for each of the decision classes. In modes e and f, each entry containing the degree of match also indicates (by parenthesized number) which of the rules for the class best matched the test event.

If the value b is instead present, a shortened confusion matrix will be reported. This will contain most of the information given in mode c, but will only list one degree of match -- the highest one found.

If the value a is instead present, a confusion matrix is generated for classes, rather than individual examples. Rows represent the intended classes, columns represent the assigned classes, and the entries represent the number of testing examples for which the denoted result occurred.

If no values a, b, or c are present, only the testing summary is provided. The summary provides the number of events correct, the number of events incorrect, the percentage of correct events, and the total numbers of conditions and rules in the ruleset.

Example testing summaries using the m, q and w options are provided in Section 4.4 along with examples of full confusion matrices. Details of the testing methods are given in Section 6.2.

trunc (yes)
An optional parameter that specifies whether rule truncation should be performed during learning. If this parameter is set to "yes", rules will be removed from the output rule set provided that the remaining rules are still consistent with the training examples based on the "weighted flexible match" criterion. The rule truncation process was designed to remove "lightweight" rules – rules that describe only a few training examples for the decision class. However, it tends to also remove some heavier rules while leaving some of the lightweight rules in the ruleset. The reason for this behavior is that the heavyweight rules removed are largely redundant – many of the examples they cover also satisfy other rules for the decision class, and typically even the examples uniquely covered by the rule lie in close proximity in the event space to the partially overlapping rules. Lightweight rules on the other hand often represent anomalous cases in remote portions of the event space.

When the trunc option is used and rules are successfully removed, AQ19 returns both the ruleset learned prior to truncation and the one created by trunc's removal of rules.

verbose (1)

Used in: Rule Learning, Knowledge Testing
Optional parameter that specifies whether the time taken by the learning and/or testing process is to be added to the output from AQ19. The verbose parameter is not solely responsible for controlling the contents of the output. The *echo* parameter controls which tables are echoed, the *wts* parameter (see below) controls the level of detail in the rule descriptions, and the *test* parameter controls the testing summary detail. The default value for the verbose parameter is 1. Legal values are:

0 No learning or testing times are echoed.

1 Learning/testing time is echoed.

wts (cpx)
Optional parameter that specifies whether AQ is to display weights with the rules it produces. The weights can give the user a gauge of the importance of individual rules or conditions based on how much they discriminate between the classes and how many of the input examples they actually applied to. Legal values are:

no — Include no weights in the output.

cpx — Two weights are associated with each complex (rule). The first weight is the total number of positive events that the rule covers (the *total* weight, denoted by "t"). The second weight is the number of events covered by this rule and no other rule in the cover (the *unique* weight, denoted by "u"). Rules for a given class will be displayed in decreasing order of the t-weight. If the rules were generated using the trunc option, a third weight will be given (the *flexible-match* weight, denoted by "f"), showing the number of events covered by no rules after truncation, but matching best to this rule using the weighted flexible match criterion. If the rules were learned in "noisy" mode, such that inconsistent rules may be present, a weight denoted by "n" will indicate the number of negative examples covered by the rule.

evt — In addition to the two weights (total and unique) calculated for each complex, a list of example indices is printed. These indices list by number, or by key field if keys were included in the events tables, the positive examples that are covered by each rule. If rule truncation was used, the examples matched flexibly to the rule are listed in a separate line.

sel — Weights are calculated for each selector (condition) in the ruleset. There are two weights associated with each selector. The first weight is the number of positive examples covered by the condition, and the second weight is the number of negative examples covered by the condition. When selector weights are shown, the conditions within a rule are displayed in decreasing order of the ratio of the first weight to the sum of both weights, i.e., the percentage of positive events covered. Otherwise the conditions in a rule are displayed in the order that their attributes (as given in the TERM portion) are given in the variables table.

all — All weights and example information is printed for each selector and complex. In other words, *all* is the union of *evt* and *sel*.

A sample parameters table is shown below. This table directs AQ to run twice. Values in the first row are the default and control the first run. Values in the second row control the second run of AQ on the same data. 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 the next section for a description of criteria tables. Parameters not present in the parameter table (e.g. "run", "mode" and "maxstar") take their default values in both runs.

Example:

```
parameters
ambig   genlevel    wts   echo   criteria
neg     mini        cpx   pvne   default
pos     gen         all   pvne   mincost
```

5.2 The criteria tables

All criteria tables other than the "default" *must* be defined. This table type is used to define a lexicographic evaluation function (LEF). An LEF evaluates a set of candidate hypotheses, using a series of preference 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 criterion. Those that qualify under that criterion are then examined according to the third one, and so forth. The LEF is used by STAR to judge the quality of each complex formed during learning. The 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 deviation from the optimal found value within each criterion.

A criteria table name consists of two parts separated by a hyphen ("-") -- the specific name, which must appear in the "criteria" column of the parameters tables (in the example above "mincost" was used), and the table identifier, "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 criterion that is to be applied at this point in the LEF. There are eleven defined criteria. From these eleven the LEF that best describes the user's rule preference is built. At least one and at most all eleven 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 criteria 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 events covered. Computationally expensive.

totvsneg (4) - maximize the ratio between the total number of positive covered examples and all negative events covered. Computationally expensive.

mincost (5) - minimize the total cost of the variables used (see Section 4.3.4).

minsel (6) - minimize the number of extended selectors (conditions).

maxsel (7) - maximize the number of extended selectors.

minref (8) - minimize the number of references (permitted values) in the extended selectors.

maxgain (9) - maximize the information gain ratio of the cover being built, based on the definition in [Quinlan, 1993].

newdifneg (10) - maximize the difference between the number of newly covered positive examples and covered negative examples. This will produce a ranking of rules equivalent to that in IREP (Fürnkranz and Widmer, 1994), RIPPER (Cohen, 1995), and PROMISE (Baim, 1982), although this is true of the latter only under circumstances of ordinary example distribution (Kaufman and Michalski, 1999).

minent (11) - minimize the entropy measure used to evaluate rules in CN2 (Clark and Niblett, 1989).

tolerance (0.00)

This mandatory column specifies the relative tolerance in the importance of this criterion. In a strict LEF (tolerance = 0) any complex not having the best (or equal) value for a criterion is immediately eliminated. This real-valued number specifies the degree of tolerance in the importance of the criterion given in the same line. As an example, suppose the best complex in a list had a value of 100 for the first defined 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 thereby allows a leniency range of 20. Any complex with a value between 80 and 100 will not be eliminated from the list of rules under consideration; instead they would make up the set of rules evaluated under the second criterion.

An example is given below. The first -criteria table given is the default. In many experiments, this criteria table will produce 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
```

5.3 The domaintypes table

The domaintypes table is used to define domains for the attributes by which the input and output events are defined. 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, size, 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 (as defined in the size column) 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. Five domain types are legal. The legal types are described below:

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 (e.g. types of food). A variable with a structured domain requires that domain be described in a structure table (Section 5.6) as well.
con	A "continuous" domain consists of real-valued numbers.

size (2)

This optional integer value specifies the number of values in the domain being defined. There is no preset limit on domain size. This value has no effect on continuous variables.

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 the criteria table, Section 4.2.3). 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. For instance, in a medical domain, it would be desirable to make a diagnosis via a blood test, rather than exploratory surgery.

To only generate rules that involved the results of such surgery when absolutely necessary, the attribute describing the results of the surgery could be assigned a prohibitively high cost, while the blood chemistry attributes would have low costs.

The domaintypes table is normally used in conjunction with the variables table and the names table. An example of both the variables table and domaintypes is given at the end of the following section.

5.4 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 domain size of a variable.

 # (1..n)

This optional column numbers the entries in the variables table. Values must be sequential integers.

name (x*n*)

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 this variable was defined. The value in the name column takes the form name.domain-name, where name is the alpha string name of the specific variable while domain-name is a more general name of the domain (if defined in the domaintypes table), or simply a repetition of the variable name.

type (nom)

This column specifies the type of the variable domain. Six 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 (e.g. types of food). A variable with a structured domain requires that domain be described in a structure table (section 5.6) as well.
con	A "continuous" domain consists of real-valued numbers.
ign	An attribute present in the data, but to be ignored during learning.

size (2)

This integer gives the number of legal values in the domain being defined. There is no limit on the domain size. The size of a structured variable is defined as the total number of nodes in the hierarchy (internal and leaf). This value has no effect on continuous variables.

cost (1.0)

This real number specifies the relative 'expense' of the domain being defined on this line. The value is used when computing the criterion mincost is used in the LEF. The default cost is 1.0. Further explanation is given in Section 4.2.4.

The following example shows domaintypes and variables tables for a computer selection task. Notice that five of the nine variables take on Boolean domains.

Example:

```
domaintypes
name type   size
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         1.0
7  processor            1.0
8  memory               100.0
9  printer.boolean      1.0
```

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

5.5 The names tables

The names tables are used to specify the legal domain values for an attribute. These must include 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 size-1 (size is defined in the variables or domains 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 table, value and name.

value (1..n)
This column must be an integer beginning with '0' and continuing sequentially up to size-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 a domain that is 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 examples of the names tables for the computer selection problem defined above. All variables that are of type “boolean” may take values “yes” and “no”. The domain “make” has the values “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
```

5.6 The structure tables

The use of structure tables is optional. A structure table is used to define a structured domain 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, in which the lowest level corresponds to the values of the variable at the lowest level of generality. Higher levels (as defined the structure table) specify parent nodes in the hierarchy of values and are used to simplify classification rules.

For instance, the hierarchy for the structured variable shape may have curve and polygon as top-level nodes, circle and ellipse as leaves under curve, and triangle and square among the leaves under polygon.

The specific name of a structure table must be the name of the domain, as specified in the name column of the domaintypes table, or if the domaintypes table is not specified, in the 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 element in the hierarchy corresponding to the value given in the "value" column of the table. If this column is included, the names used in this column may appear in classification rules instead of the values named in the names table or the events tables.

value

This mandatory integer entry specifies a parent node in the hierarchy. This node will be defined as the parent of the nodes specified in the subvalues column. If this value is a subvalue of some other values, the row in which this value is declared as a parent must appear before any rows in which it is listed as a subvalue (e.g. entries must be given in a bottom-up order). This integer value must always be greater than any of the subvalues in the following column.

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 ".." to indicate ranges. These numbers correspond to values as defined in the names table of the variable or previous rows of the structure table.

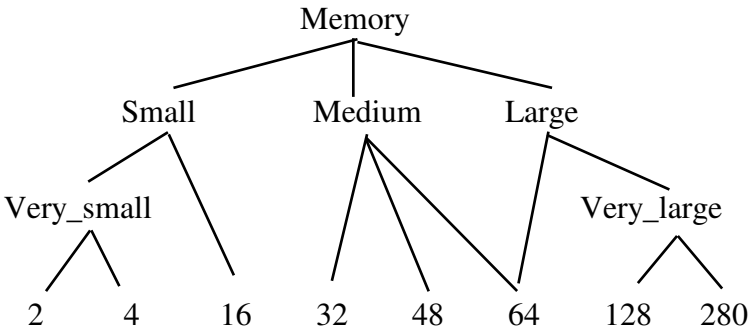
The hierarchical graph below shows an example of a structured domain for the variable "memory". Note that the same node (64 in this example) may be shared by multiple parents ("medium" and "large"). Below are the input tables to define such a structure.

Note that for the variable "memory" a names table must first be defined, because a domain of all values between 2 and 280 would be inefficient and might cause inaccurate rules. It should also be noted 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
  8      very_small
  9      small
 10      medium
 11      very_large
 12      large
```

```
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
```



5.7 The inhypo and outhypo tables

The inhypo tables are optional and are used to input rules for incremental learning (i.e., learning by modifying prior hypotheses, rather than from examples alone), for decision structure generation, for standalone rule testing, and for knowledge visualization. (*Inhypo* stands for "input hypothesis," and *outhypo* for "output hypothesis." The difference comes from whether they were provided by the user or directly from an earlier AQ19 output file). The specific name of each inhypo table must match the name of one of the decision classes. The rules input in an inhypo table have three possible roles. In the first, when there is at least one events table specified, the input (inhypo) rules are used as initial covers for incremental learning. If no events tables or tevents tables are present, the input rules are treated as events for rule optimization. If no events tables are present, at least one tevents table is present, and the testonly parameter is "yes", the input rules are used for testing alone.

If inhypo rules for different classes intersect in the event space, then their intersection is treated as determined by the "ambig" parameter in the parameters table (Section 5.1).

There are two columns in the inhypo table.

 # (1..n)
This mandatory column associates a number with each complex (rule) in the ruleset for the class being described by this inhypo table. 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 ([])
rule ([])

This mandatory column specifies the VL_1 rule. A complex (rule) is presented as a conjunction of selectors (conditions), each enclosed in square brackets. The syntax of complexes and selectors are defined in Section 2. Extended selectors (in which more than one variable sharing the same domain are given together in a condition) may be used, with the variables being delineated by periods.

The degrees of match determined by rule testing are in some of the testing modes affected by the number of training examples covered by a rule. In order to pass that information to the rule testing module in testonly mode, rules' total and uniquely covered weights may be included in

parentheses at the end of those rules. This is optional information that will have no effect on incremental learning or rule optimization.

Below are examples of an inhypo table:

Example:

```
Under1000-inhypo
  #  cpx
  1  [floppies=0]    (t:8, u:5)
  2  [cd_rom.parallel_cpu = no]    (t:4, u:1)

From1000_4000-inhypo
  #   cpx
  1   [floppies = 1,2][memory > 16]    (t:12, u:12)
  2   [floppies > 3][memory >= 4]    (t:3, u:3)
```

5.8 The events and tevents tables

While events and tevents tables have identical structure, the examples contained in events tables are used for learning and those in tevents tables are used in testing. Both types of 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 three column types: 1) row number, 2) Key and 3) 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.

Key ()
This optional column permits the user to include an identifier for the example in alpha form. This feature is useful when the wts parameter is set to ‘all’ or ‘evt’ (Section 6.3.1). It appears like any other variable column, but in its role as strictly an identifier, its values are not used in learning.

Weight (1)
This optional column allows an example that occurs more than once to be inputted just once; the number in this column serves as a count of the number of instances of such an example.

variables (x1..xn)
This definition 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 a set 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
```

5.9 The children tables

The optional children tables define the hierarchical ordering of the values of the decision variable (i.e., the different classes) in cases in which the decision variable is structured. 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 child node being defined, i.e., the subclass of the class by which the table is titled.

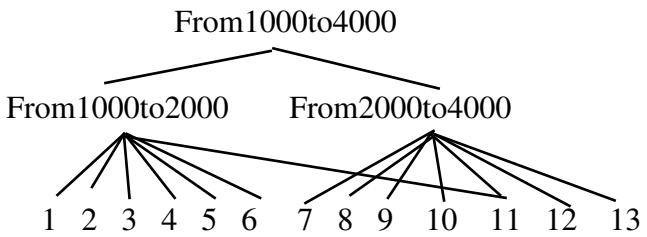
events
This mandatory column is a list of indices of events belonging to the parent node (the node on which the table's name is based) that are examples of this child node. The list of indices may use commas (1,2,3,4) or ranges (1..4). The events are numbered in the order they appear in the parent node's events table.

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

The example below defines the two classes "From1000to2000" and "From2000to4000" which are subclasses 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".

Example:

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



5.10 The varsel table

The optional varsel table allows the user to specify instructions for AQ19 to reduce the attribute set prior to learning. By selecting only the more task-relevant attributes, AQ19 can learn more quickly, and often will generate clearer, more useful knowledge. Attribute selection is based on the PROMISE algorithm (Baim, 1982) and the modifications to it (Kaufman, 1997) to allow for rule-oriented output.

The varsel table consists of two columns:

varbasis (max)
This alpha column specifies the means by which the attribute selection engine will rate the different attributes. The user may choose between two values:

max — This value biases the selection process toward attributes that will work well in decision rules or other declarative representations. The preferred attributes will have values that discriminate well among the classes.

avg — This value biases the selection process toward attributes that will work well in decision trees/structures or other procedural representations. The preferred attributes will have strong overall discriminatory power given a distribution of values similar to that found in the training data.

cutoff (t50)
This column consists of a string specifying the basis for retention or removal of attributes. The string consists of a letter, either 'a' or 't', followed by a number.

If the first character of the string is 'a', AQ19 will retain the best *n* attributes, where *n* is the number specified in the remainder of the cutoff string. Obviously, *n* must have a value of at least 1, and less than the original number of attributes in the dataset.

If the first character of the cutoff string is 't', the number following it sets a threshold attribute promise value that attributes must reach to be retained. The range of the promise levels is 0-100, with 0 signifying no discriminatory utility and 100 signifying perfect discrimination.

In the example below, the user has specified attribute selection based on the attributes' promise for performance in decision rules. Only the best six attributes will be retained.

Example:
```
varsel
  varbasis    cutoff
  max          a6
```

6 TESTING RULESETS

6.1 Objective

The Testing module is used to evaluate the performance of a collection of rulesets (a rule base). This new module incorporates the functionality of the separate testing program known as ATEST (Reinke, 1984). By sharing the input format with AQ19, it becomes a convenient operator in the AQ19 package, allowing the user immediate analysis of discovered rules.

6.2 Testing Methods

In this module the fundamental operation is the calculation of the degree of match between a rule and an example in the form of a vector of attribute values. The calculated value is called a degree of confidence. The calculation of confidence varies with the method of evaluation selected by the user in the test parameter in the parameters table. Full descriptions of these methods are given below based on the following test parameters:

m INLEN, or rform mode used for testing. This mode is named after INLEN in which it is used (Michalski et al, 1992). In this method of evaluation:

> For every testing example, evaluate each class ruleset (cover) against the testing example.
>
> If a match exists between one or more covers and the example:
>
> > For each cover, confidence is the *probabilistic sum* of confidences for each rule in the cover, where the probabilistic sum of occurrences A and B is P(A) + P(B) - (P(A) * P(B)). A rule's confidence calculation depends on the source of the evidence for that rule.
> >
> > If the rule was learned from training examples provided in the current learning session, then the rule's confidence is the percentage of examples of its class covered by that rule (i.e. a heavier rule or one that better describes a large share of the training examples will have a higher confidence when matching a new testing example than a match to a 'lighter' rule.)
> >
> > Else:
> >
> > The rule's confidence is the ratio of testing examples of that class covered by that rule to the total number of testing examples of that class covered by any rule for that class (i.e. in the absence of training examples, a 'stronger' rule is one that matches more of the testing examples).
>
> Else (if no match exists between a class ruleset and the testing example):
>
> > A class confidence is a probabilistic sum of confidences for each rule in the cover. A rule's confidence is calculated as the ratio of the number of conditions in the rule matching the example to the total number of conditions in the rule.

q hgform evaluation method first described in (Michalski and Chilausky, 1980). In this method of evaluation:

> Evaluate each cover against all testing examples.
>
> For each example:
>
> If a match exists between a rule for a given class and the example:
>
> > For each class, the degree of match (confidence) is the number of training examples of the class covered by any of the rules that match the testing

example. This value is normalized by dividing all confidence values by the maximum calculated confidence.

Else:

For each class, the confidence is the probabilistic sum of confidences for each rule in the cover. A rule's confidence is calculated as the ratio of the number of conditions in the rule matching the example to the total number of conditions in the rule.

w igform evaluation method described by Michalski (1986). In this form of evaluation:

Evaluate each class ruleset against all testing examples.

For each example:

If a match exists between a rule in a cover and the example:

For each class, the degree of match (confidence) is the probabilistic sum of confidences for each rule in that class. A rule's confidence is calculated as the ratio of the number of events covered by the rule to the total number of all events (for all classes). This evaluation scheme assumes the distribution of training examples in each class is characteristic of the distribution of examples in the universe of possible examples.

Else:

For each class, the confidence is the probabilistic sum of confidences for each rule in the cover of that class. A rule's confidence is calculated as the product of the confidences for each condition in the rule weighted by the percentage of all training examples covered by the rule. A condition's confidence is 1 if it covers the testing example otherwise it is a value between 0- and 1 proportional to the ratio of the number of possible values satisfying the condition to the full domain size.

e Equally weighted flexible matching evaluation method. In this form of evaluation:

A class confidence is the maximum confidence for the rules in the class cover. A rule's confidence is calculated as the ratio of the number of conditions in the rule matching the example to the total number of conditions in the rule. To break ties in determining the class best matched, the class will be determined by which rule covered the most training examples.

f Weighted flexible matching evaluation method (not available in testonly mode). In this form of evaluation:

A class confidence is the maximum confidence for the rules in the class cover. A rule's confidence is calculated as follows:

For each condition, determine its informativeness weight, which is defined as the number of training examples of the decision class that satisfy the condition divided by the total number of training examples that satisfy it.

Normalize by dividing by the sum of all of the informativeness weights for the conditions comprising the rule. A rule's confidence is determined by taking the sum of the normalized informativeness weights of the conditions in the rule matching the testing example.

To break ties in determining the class best matched, the class will be determined by which rule covered the most training examples.

Example testing summaries are provided in section 4.3.2, along with examples of full confusion matrices using the m, q and w options.

7 AQ19'S PREDECESSORS: A LITTLE HISTORY FOR THE CURIOUS

The underlying framework of AQ19 is the STAR methodology (Michalski, 1969; 1983), and the A^q algorithm for solving the general covering problem (Michalski, 1969); the name stands for "**a**lgorithm for **q**uasi-optimal solutions". AQ19 can, however, execute a variety of modifications of the basic algorithm, in order to fit the learning process to the problem at hand. For example, it can take into consideration the level of noise in the data, and the size and type of data, optimize rulesets according to many different preference criteria, and produce the different types of output descriptions (intersecting, disjoint, ordered, maximally general, maximally specific, and/or optimized).

The termination criterion depends on the control parameters. The default parameters assume that data are large and may have some errors, so the system seeks rulesets that optimize the description quality criterion (Kaufman and Michalski, 1999; Michalski, 2001).

AQ19 is an immediate descendant of AQ18 (Kaufman and Michalski, 2000), which is an immediate descendant of AQ15c (Wnek et al, 1995), a C-based implementation of AQ15 (Hong, Mozetic and Michalski, 1986). AQ15 was an enhancement of the GEM (Generalization of Examples by Machine) program, written by Bob Stepp and Mike Stauffer. GEM was written from scratch on the Cyber 175, rather than as a modification of the AQ7-AQ11 series of programs (Michalski, 1969; Michalski and Larson, 1975; 1978; 1983).

AQ15c included a number of new features that extended the functionality of previous versions. One change from previous versions is the language in which it is written. AQ15c was ported to ANSI-C. The most important advantage of this change is that it reduced the need for 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 forced upon the programmers a complex way of handling sets, and limited attributes in the data sets to 58 values. The ANSI-C version had no preset limitations on the number of variables, the number of values per variable, and the number of classes. In addition, the ANSI-C version proved to be on average six times faster than the previous implementation, and provided better error diagnostics.

Another important result of the port from Pascal to ANSI-C was the ability to easily port the program to different platforms. The AQ15c version was ported to Sun Solaris, IBM-compatible, and Apple platforms. More platforms can be supported in the future with no change in the underlying source code because of the use of ANSI-C.

AQ18 featured many innovations described elsewhere in this report, including the introduction of no-loss rule truncation, pattern discovery mode for learning rulesets that may not be complete

and consistent, in-program attribute selection, two new means of handling ambiguity, three new LEF criteria, full flexible matching modes for ATEST, and a reimplementation of uniclass learning for the characterization of a single class of examples.

Other recent AQ programs have explored new capabilities from the AQ15 standard as well. AQ16 (Poseidon) implemented a method for modifying AQ-learned rules by optimizing rulesets, using operators of specialization (rule truncation and reference reduction) and generalization (reference extension). This method was based on ideas of two-tiered concept representation (Bergadano et al, 1992). The AQ17 family consists of three different programs: AQ17-DCI, a program for data-driven constructive induction (Bloedorn and Michalski, 1998); AQ17-HCI, a program for hypothesis-driven constructive induction (Wnek and Michalski, 1994); and AQ17-MCI, a program for multistrategy constructive induction (Bloedorn, 1996).

REFERENCES

Baim, P., "The PROMISE Method for Selecting Most Relevant Attributes for Inductive Learning Systems," *Reports of the Intelligent Systems Group*, ISG 82-1, UIUCDCS-F-82-898, Department of Computer Science, University of Illinois, Urbana, September, 1982.

Bergadano, F., Matwin, S., Michalski, R.S. and Zhang, J., "Learning Two-tiered Descriptions of Flexible Concepts: The POSEIDON System," *Machine Learning*, Vol. 8, No. 1, pp. 5-43, January 1992.

Bloedorn, E.E., "Multistrategy Constructive Induction," Ph.D. Dissertation, School of Information Technology and Engineering, *Reports of the Machine Learning and Inference Laboratory,* MLI 96-7, George Mason University, Fairfax, VA, 1996.

Bloedorn, E. and Michalski, R.S., "Data-Driven Constructive Induction," *IEEE Intelligent Systems*, Special issue on Feature Transformation and Subset Selection, pp. 30-37, March/April, 1998.

Clark, P. and Niblett, T., "The CN2 Induction Algorithm," *Machine Learning* 3, pp. 261-283, 1989.

Cohen, W., "Fast Effective Rule Induction," *Proceedings of the Twelfth International Conference on Machine Learning*, Lake Tahoe, CA, 1995.

Fürnkranz, J. and Widmer, G., "Incremental Reduced Error Pruning," *Proceedings of the Eleventh International Conference on Machine Learning*, New Brunswick, NJ, 1994.

Hong, J., Mozetic, I., Michalski, R.S., "AQ15: Incremental Learning of Attribute-Based Descriptions from Examples, the Method and User's Guide," *Reports of the Intelligent Systems Group*, University of Illinois at Urbana-Champaign, ISG 86-5, May, 1986.

Kaufman, K.A., "INLEN: A Methodology and Integrated System for Knowledge Discovery in Databases," Ph.D. Dissertation, School of Information Technology and Engineering, *Reports of the Machine Learning and Inference Laboratory*, MLI 97-15, George Mason University, Fairfax, VA, 1997.

Kaufman, K.A. and Michalski, R.S., "Learning in an Inconsistent World: Rule Selection in AQ18," *Reports of the Machine Learning and Inference Laboratory*, MLI 99-2, George Mason University, Fairfax, VA, 1999.

Kaufman, K.A. and Michalski, R.S., "The AQ18 System for Machine Learning: User's Guide," *Reports of the Machine Learning and Inference Laboratory*, MLI 00-3, George Mason University, Fairfax, VA, 2000.

Kerber, R., "Chi-Merge: Discretization of Numeric Attributes," *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, San Jose, CA, pp. 123-127, 1992.

Michalski, R.S., "On the Quasi-Optimal Solution of the General Covering Problem," *Proceedings of the V International Symposium on Information Processing (FCIP 69)*, Bled, Yugoslavia Vol. A3 (Switching Circuits), pp. 125-128, 1969.

Michalski, R.S., "Variable-Valued Logic and Its Applications to Pattern Recognition and Machine Learning," Chapter in Rine, D.C. (Ed.), *Computer Science and Multiple-Valued Logic Theory and Applications*, North-Holland Publishing Co., pp. 506-534, 1975.

Michalski, R.S, "A Theory and Methodology of Machine Learning," in Michalski, R.S, Carbonell, J.G. and Mitchell, T.M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Tioga Publishing Company, 1983, pp. 83-134.

Michalski, R.S., "Natural Induction: A Theory and Methodology," *Reports of the Machine Learning and Inference Laboratory*, MLI 01-1, George Mason University, Fairfax, VA, 2001.

Michalski, R.S., and Chilausky, R.L., "Learning By Being Told and Learning From Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *Policy Analysis and Information Systems,* Vol. 4, No. 2, 1980.

Michalski, R.S. and Kaufman, K.A., "Data Mining and Knowledge Discovery: A Review of Issues and a Multistrategy Approach," in Michalski, R.S., Bratko, I. and Kubat, M. (Eds.), *Machine Learning and Data Mining: Methods and Applications*, London: John Wiley & Sons, pp. 71-112, 1998.

Michalski, R.S., Kerschberg, L., Kaufman, K. and Ribeiro, J., "Mining for Knowledge in Databases: The INLEN Architecture, Initial Implementation and First Results," *Journal of Intelligent Information Systems: Integrating AI and Database Technologies*, Vol. 1, No. 1, August 1992, pp. 85-113.

Michalski, R.S. and Larson, J., "AQVAL/1 (AQ7) User's Guide and Program Description," Report No. 731, Department of Computer Science, University of Illinois, Urbana, June 1975.

Michalski, R.S. and Larson, J., "Selection of Most Representative Training Examples and Incremental Generation of VL_1 Hypotheses: The Underlying Methodology and the Description of Programs ESEL and AQ11," Report No. 867, Department of Computer Science, University of Illinois, Urbana, May 1978.

Michalski, R.S. and Larson, J., "Incremental Generation of VL_1 Hypotheses: The Underlying Methodology and the Description of Program AQ11," ISG 83-5, UIUCDCS-F-83-905, Department of Computer Science, University of Illinois, Urbana, January 1983.

Michalski, R.S., Mozetic, I., Hong, J., and Lavrac, N., "The Multi-Purpose Incremental Learning System AQ15 and its Testing Application to Three Medical Domains," *Proceedings of AAAI-86*, pp. 1041-1045, Philadelphia, PA, 1986.

Moczulski, W., "Inductive Learning in Design: A Method and Case Study Concerning Design of Antifriction Bearing Systems," in Michalski, R.S., Bratko, I. and Kubat, M. (Eds.), *Machine Learning and Data Mining: Methods and Applications*, London: John Wiley & Sons, 1998.

Quinlan, J.R., *C4.5: Programs for Machine Learning*, Morgan Kaufmann, Los Angeles, CA, 1993.

Reinke, R.E., "Knowledge Acquisition and Refinement Tools for the ADVISE Meta-Expert System," University of Illinois at Urbana-Champaign, Master's Thesis, 1984.

Thrun, S.B., Bala, J., Bloedorn, E., Bratko, I., Cestnik, B., Cheng, J., De Jong, K., Dzerowski, S., Fahlman, S.E., Hamann, R., Kaufman, K., Keller, S., Kononenko, I., Kreuziger, J., Michalski, R.S., Mitchell, T., Pachowicz, P., Vafaie, H., Van de Velde, W., Wenzel, W., Wnek, J., Zhang, J., (1991). "The MONK'S Problems: A Performance Comparison of Different Learning Algorithms," *(revised version)*, Carnegie Mellon University, Pittsburgh, PA, CMU-CS-91-197.

Wnek, J., Kaufman, K., Bloedorn, E. and Michalski, R.S., "Inductive Learning System AQ15c: The Method and User's Guide," *Reports of the Machine Learning and Inference Laboratory*, MLI 95-4, George Mason University, Fairfax, VA, 1995.

Wnek, J. and Michalski, R.S., "Hypothesis-driven Constructive Induction in AQ17-HCI: A Method and Experiments," *Machine Learning*, Vol. 14, No. 2, pp. 139-168, 1994.