Reports

Machine Learning and Inference Laboratory

AQ21 User's Guide

Janusz Wojtusiak

MLI 04-3
P 04-5
September, 2004
Updated: September, 2005

School of Computational Sciences

George Mason University

AQ21 USER'S GUIDE

Janusz Wojtusiak
Machine Learning and Inference Laboratory
George Mason University

Fairfax, VA 22030-4444, USA
jwojt@mli.gmu.edu
http://www.mli.gmu.edu

Abstract

AQ21 is a multitask machine learning and data mining system for attributional rule learning, rule testing, and application to a wide range of classification problems. One of its distinctive features is that it strives to perform natural induction, that is, it seeks inductive hypotheses that are not only accurate but also easy to understand and interpret. Although the system provides the user with a large number of control parameters, they all can be omitted if one wants to run it in default mode. The parameters control the mode of learning (theory formation vs. pattern discovery), the degree of generality of learned rules, and a range of preference criteria for selecting candidate rules that tailor the learning process to the given problem. AQ21 includes event and episode classification programs (ATEST and EPIC) that give the user control over the testing and application of the learned rulesets to the task at hand.

Keywords: AQ learning, machine learning, natural induction, data mining and knowledge discovery, knowledge mining, multistrategy learning, learning form examples.

Acknowledgments

Author would like to express his gratitude to all people who helped in development of AQ21 system and this manual. Professor Ryszard S. Michalski, Director of the Machine Learning and Inference Laboratory, continuously supervised work on AQ21 and provided the theory needed for development of the AQ21 system. He also reviewed this manual and provided constructive comments. Dr. Kenneth Kaufman helped prepare and correct the manual. He also provided information about the AQ methodology and previous implementations of AQ (including AQ19). Dr. Bartlomiej Sniezynski and Jarek Pietrzykowski provided a number of comments on this manual and the AQ21 program.

AQ21 source code includes significantly modified, debugged, improved, and extended with many new features source code of AQ20 program developed in the Machine Learning and Inference Laboratory.

Research activities in the Machine Learning and Inference Laboratory are supported in part by the National Science Foundation Grants No. IIS 9906857 and IIS 0097476, and in part by the UMBC/LUCITE #32 grant.

This report is an extended and corrected version of the technical report MLI 04-3.

.

1 INTRODUCTION

Large amounts of data produced in the current world need to be analyzed and transformed into knowledge. Research in *Data Mining* is focused on extraction of useful knowledge from large amounts of data. The recently introduced field of *Knowledge Mining* emphasizes the use of prior knowledge and the understandability of learned descriptions according to the idea of *natural induction*. Usefulness of the learned knowledge means not only that the knowledge should be accurate and free of errors, but also that it should be presented in a human-oriented way. This requires the use of a language for learning that is human-oriented and at the same time allows the inductive learning process to be effectively performed. Such a language is *Attributional Calculus* (Michalski, 2004), a subset of which is used in the AQ21 program.

AQ21 is a machine learning program whose learning module is based on an integration of a simple version of the A^q algorithm (Michalski, 1969) for solving the general covering problem with variable-valued logic (Michalski, 1972). The central part of the algorithm focuses on the concept of a *star* (a set of maximal generalizations of a given positive example) and its generation. Given sets of positive and negative examples of some concept, the A^q algorithm learns a complete and consistent description of that concept. Several modifications to the algorithm allow that the learned descriptions may not be complete and/or consistent (for example *mode* parameter described in Section 8.3).

The AQ21 program is the latest member of the AQ family of programs for rule learning and testing. Among the best known AQ programs are AQVAL/1 (AQ7) (Michalski and Larson, 1975), AQ11 (Michalski and Larson, 1983), AQ15c (Wnek et al., 1995), AQ17-DCI (Bloedorn and Michalski, 1991), AQ17-HCI (Wnek and Michalski, 1994), AQ17-MCI (Bloedorn et al., 1993), and AQ19 (Michalski and Kaufman, 2001).

There are a number of other machine learning programs, among which are well-known decision tree learners such as ID3 and C4.5 (Quinlan, 1993), and rule learners such as CN2 (Clark and Niblett, 1989) and RIPPER (Cohen, 1995). The CN2 program is based on some aspects of AQ learning. There are also toolboxes that combine a number of machine learning programs e.g., WEKA (Witten and Frank, 1999). The INLEN system (Kaufman, 1997), and a much more extended VINLEN system (still under development; Kaufman and Michalski, 2003) integrate a range of machine learning and other programs to provide an environment for multistrategy data exploration, visualization, decision making and problem solving.

AQ21 comes in two versions, (a) as a stand-alone program, and (b) integrated as an operator of VINLEN. AQ21 is also integrated with iAQ program intended to demonstrate the concept of natural induction, and to support teaching and research in data mining, and computational learning and discovery (Michalski and Pietrzykowski, 2004)

2 HARDWARE AND SOFTWARE REQUIREMENTS TO RUN AQ21

AQ21 can be run on Linux platform (Red Hat 7.3 or other) and Windows environment (Windows 95 or newer) with free RAM memory of at least 16 MB. To illustrate the amount of time and memory needed to execute the program, Table 1 gives information about the program performance on two problems, one relatively simple (about 7000 training examples defined by 32 binary variables and characterizing a single concept and its negation), and one more complex (over 8700 training examples defined by 32 multi-valued attributes, and characterizing 10 classes). The first problem comes from a medical domain and the second problem comes from a user modeling domain. In both cases AQ21 was executed on a Dell PowerEdge 4600 Server equipped with Intel Xeon 2 GHz processors, 1GB of RAM, and running the Red Hat Linux 7.3 operating system. For many applications much slower machines with less RAM can be used.

Problem	Number of training events	Number of testing events	Number of classes	Number of attributes	Learning Time	Testing Time	Number of rules learned	Max. RAM memory used
1	5000	2000	2	32 (all binary)	24 sec	< 1 sec	5	3.8 MB
2	9041	4139	10	24 (mostly linear, 5-10 values per attribute)	458 sec	12 sec	3262	9.4 MB

Table 1: Example execution times of AQ21

We also successfully tested AQ21on much larger problems, e.g., a problem with over 100,000 examples and 30 attributes, and on a problem with 200 examples and 500 attributes.

Although a study of complexity of the AQ algorithm is out of scope of this manual, it should be mentioned that time and memory complexity of the AQ21 program depends on number of examples, number of attributes, and number of values in attributes' domains. Although the program has no specific limit on any of the above factors, in the case it runs too slow, the User should apply some method for data reduction (to reduce the number attributes, the number of examples, or the domain sizes).

3 HOW TO START WORKING WITH AQ21

3.1 Running AQ21 and list of its the most important parameters

To run AQ21 issue a command:

aq21 input-filename

Input file, whose name is given as only parameter of execution, is a plain text file that consists of eight components to be specified in the order presented below. Components: ***Attributes*** and ***Runs*** are always mandatory. For rule learning, component ***Events*** is also mandatory, and for rule testing, one of components, ***Events*** or ***Testing_events***, is also mandatory. The remaining components are optional. Each component starts with a section name and is followed by a text enclosed in braces"{ }". The component names are keywords in the program and have to be typed exactly as stated below. In this manual, all key words are typed in bold and italics for emphasis, but in the actual data specification, these key words are typed as any regular text.

1. ***Problem_description*** (Optional)

   This component allows the user to write comments about learning and testing data and problem to be solved. It is ignored by AQ21 during learning or testing, but the text between braces is copied to the program output. Example:

Problem_description *{* *This is an example description. User can type here any text that will be displayed in the program output.* *}*	***Comment:*** Any text can be written between braces {} that does not include symbols "{" and "}"

2. ***Common_domains*** (Optional)

   This component allows the user to define domains (sets of legal values) to be assigned to attributes in the ***Attributes*** component. Defining these domains here is particularly useful when several attributes have the same domain. An alternative way to define attribute domains is to define them directly in ***Attributes***. For more detail on this topic see Section 5. Example:

Common_domains *{* color nominal {r, y, b, g, w} length continuous 0, 500 size linear {s, m, l} shape structured {curved, quadrangle, circle, ellipse, square, rectangle} {circle --> curved ellipse --> curved square --> quadrangle	***Comments:*** Domain "color" is defined as *nominal* and has values: r, y, b, g, w. Domain "length" is defined as *continuous* and takes values from the range between 0 and 500. Domain "size" is defined as *linear* with values ordered as s < m < l. Domain "shape" is defined as structured. In the first braces all possible values are listed, and in the second braces relations are listed. In a pair "value1 --> value2", value2 is a parent of value1 in the generalization

`rectangle --> quadrangle }` `position continuous` `}`	hierarchy. Domain "position" is defined as *continuous*, and ranges over the interval from the minimal to the maximal real number representable on the computer.

List of all domain types and their definitions in AQ21 input file:

nominal	{comma separated list of values}
linear	{comma separated list of values} (alternative name: ordinal)
structured	{comma separated list of values} {list of relations}
continuous	[upper bound, lower bound]
integer	[upper bound, lower bound]
discretized_continuous	{comma separated list of discretization points}

3. ***Attributes*** (Mandatory)

   The section that defines attributes in the input data table. Each definition of an attribute in the section corresponds to one column in the input data, in the order they are presented (columns are counted form left to right). For details please refer to Section 6 of this manual. Example:

Attributes `{` `x1 shape` `x2 nominal {r, s, t}` `x3 shape` `x4 color` `x5 continuous -10, 2000` `user nominal {1, 2, 3}` `}`	***Comments:*** Attributes x1 and x4 have the domain "shape" defined in ***Common_domains***. Attribute x2 is of nominal type and its domain is {r, s, t}. Attribute x4 has domain "color" defined in ***Common_domains***. *Attribute user is of nominal type and its domain is {1,2,3}*

4. ***Runs*** (Mandatory)

   The component contains global learning parameters and subcomponents, called *runs*, which define tasks to be performed by a single run of the program. Each *run* starts with its single name identifying the run, e.g.,"my_run1", followed by parameters enclosed in braces. The only parameter needed for learning is the rule *consequent* defined in each run. Even if the program is used only for rule testing, this component needs to be specified. For details on defining *runs* please refer to Section 8 of this manual. Example:

Runs `{` `Ignore_attributes = x3, x5` `Events_precentage = 0.7` `my_run1` `{` `Consequent = [x2 = r][user = 2]` `}` `another_run` `{`	***Comments:*** Global Parameters defined before the first run. In this case two global parameters are defined: *Ignore_attributes* and *Events_precentage*. Each run starts with its unique name, here "my_run1" and "another_run", followed by list of parameters written in braces. The only mandatory parameter is rule *Consequent* that defines class of positive examples for this run. Thus, run 1 will produce rules with *Consequent* "[x2 = r][user =2]"

`  Mode = TF` `  Consequent = [user = *]` `  }` `}`	If an'*' is used in the *Consequent*, program will automatically create classes for all possible values of the attribute.

List of the most important global parameters and their definitions in AQ21 input files:

Ignore_attributes	=	comma separated list of attributes
Counting_attribute	=	*true*|*false*
Cross_valudation	=	0, 1, …. number of events -1
Attribute_selection_method	=	*PROMISE* | *Gain_Ratio* | *None*
Attribute_selection_threshold	=	value from range [0, 1]
Attribute_selection_tolerance	=	value from range [0, 1]
Output_attributes	=	comma separated list of attributes

List of the most important run-specific parameters and their definitions in AQ21 input files:

Consequent	=	[attribute = *] (learn for all values) OR
Consequent	=	[attribute = v] (learn for one value) OR
Consequent	=	[attribute = v1,v2,v3] (selector is consequent) OR
Consequent	=	Complex (any attributional complex)
Mode	=	*TF* | *PD* | *ATF* | *TEST*
Ambiguity	=	*IgnoreForLearning* | *IncludeInPos* | *IncludeInNeg* | *IncludeInMajority* | *DisplayAmbiguities*
Define attribute_name	=	A-rule defining attribute
Exceptions	=	true | false
Ignore_attributes	=	comma separated list of attributes
LEF_PS | LEF_STAR | LEF_SORT		{list of criteria and tolerances}

5. ***Tests*** (Optional)

   This component defines the testing of learned or input (predefined) hypotheses. The user can use testing algorithms such as ATEST and EPIC and their modifications. For details please refer to section 9 of this manual. Example:

Tests `{` `  my_test1` `  {` `  Method = atest` `  Threshold = 0.5` `  }` `}`	***Comments:*** Each test starts with its unique name followed by list of parameters written in braces. In this case, only one test is defined, and its name is "my_test1". It uses ATEST event testing method with acceptance threshold 0.5.

List of the most important testing parameters and their definitions in AQ21 input files:

Method	=	*ATEST* | *EPIC* | *EPIC_RB* | *EPIC_P*
Evaluation_of_selector	=	*strict* | *flexible*
Evauiation_of_conjunction	=	*strict* | *coverage_ratio* | *selectors_ratio* | *flexible* | *min* | *min_w* | *prod* | *avg* | *avg_w*

Tolerance	=	value from range [0, 1]
Threshold	=	value from range [0, 1]
Full_report	=	*true* | *false*

6. ***Input_hypotheses*** (Optional)

   This consists of hypotheses (rulesets) used for testing. The rulesets can be either a result of previous program execution or manually created hypotheses. This component is specified ONLY when User wants to test already known hypotheses. For each decision class (defined by the consequent) a separate input hypothesis section needs to be specified. For details please refer to section 9.4 of this manual. Example:

Input_hypotheses *my_run1* *{* *[user = user1]* *<-- [x3 = r, s][x5 = r]: p = 10, n = 0* *<-- [x1 = r] : p = 10, n = 2* *}*	***Comment:*** Name (in this case my_run1) is a valid run name specified in the ***Runs*** section. Consequent of rules in the hypothesis. List of rules, each rule starting with a "<--" symbol.

7. ***Events*** (Optional)

   This component specifies training and/or testing events for the program. This section is optional in the sense that it is skipped, if the user specifies events in the *Testing_events* component or in a separate file indicated in the command line (see Section 3.2).

Events *{* *s, s, r, r, r, 1* *s, s, s, r, b, 1* *r, t, s, r, r, 2* *r, r, t, r, g, 2* *r, r, s, r, g, 2* *}*	***Comment:*** Events are specified as rows of a data table. Each of the events is a comma separated list of values of attributes in the order defined in the ***Attributes*** component. There is no comma after each event, and the last value in the event must be followed by new line

8. ***Testing_events*** (Optional)

   The section that specifies events used by the AQ21 testing module. The testing events can be also selected from training events or provided in a separate file.

Testing_events *{* *s, s, s, r, r, 1* *s, s, t, g, r, 1* *r, s, r, r, g, 2* *r, r, s, r, g, 2* *r, s, t, r, g, 2* *}*	***Comment:*** Data table is a list of testing examples (events). Each of the events is a comma separated list of values of attributes in order defined in the ***Attributes*** component. There is no comma after each event, and the last value in the event must be followed by new line.

Examples of input files are presented in Section 4, and also in Appendix A. Templates of the AQ21 input files for the most common problems are presented in Appendix C. Detailed description of structure of AQ21 input files and compatibility with C4.5 input files is presented in Section 4.

3.2 Advanced command line parameters

It was mentioned above that to execute AQ21 the user needs to provide parameters in the command line. If run without any parameters AQ21 displays information about programs and brief description about its parameters. At least one parameter is needed for learning or testing, which is the name of the input file with the problem definition and the examples for learning and/or testing. Such case was described in Section 3.1 and is shown in Example 1. In general, AQ21 is invoked by typing in the command line:

`aq21` *`command-line-parameters`*

In the simplest case, when the input file consists of problem definition and examples, the invoking command is (see Examples 2 and 3 below):

`aq21 parameters-events-`*`filename`*

It is also possible to provide problem definition, learning and testing examples in separate files:

`aq21` *`parameters–filename training-events-filename testing-events-filename`*

The parameters file is used to define AQ21 parameters (attributes, runs, tests, etc.). The training events file consists of a list of examples used for training, and a testing events file consisting of a list of testing examples may also be specified.

Example 1: *aq21 input1.aq21*	Both: parameters and events must be present in the *input1.aq21* file.
Example 2: *aq21 input2.aq21 data1*	Parameters are defined in *input2.aq21* file and data for training and/or testing is defined in *data1* file.
Example 3: *aq21 input3.aq21 data1 data2*	Parameters are defined in *input3.aq21* file, training data is defined in *data1* file, and testing data is defined in *data2* file.

In case when input file, training data file and testing data file have the same stem in the name it is possible to use ***–F*** <name> option where <*name*>*.aq21* is a AQ21 parameter file, <*name*>*.data* is a training data file, and <*name*>*.test* is an optional testing data file.

Example: *aq21 -F input*	Parameters are specified in *input.aq21* file, data is specified in *input.data* file. If *input.test* file exists, testing events will be loaded from it.

It is also possible to read input files in C4.5 format using ***–f*** option. Similarly to ***–F*** option, program loads parameters and data from respectively <name>.names, <name>.data, and <name>.test files. For details of c4.5 file format please refer to Section 4.2.

Example: `aq21 -f input`	Attributes in c4.5 format are specified in *input.names* file, data is specified in *input.data* file. If *input.test* file exists, testing events will be loaded from it.

By default AQ21 does not display events in the output. To display the training and testing events section ***–events*** parameter should be used. Please note that in such case, when datasets are large, the output will also be large.

Example: `aq21 -events input.aq21`	Parameters and events are loaded form *input.aq21* file. Program displays in its output events loaded from the input file (***Events*** section).

The AQ21 can write its output to a file whose name is specified by user. To use this feature
–o <*filename*> option should be used as shown in the example below. Format of the file is identical as format of output displayed to screen.

Example: `aq21 input.aq21 -o output.aq21`	Parameters and events are loaded from *input.aq21* file. Output is written into *output.aq21* file.

The ***–r*** option overwrites random seed specified in the input file with one generated automatically using the system time. For details of random seed usage please refer to Section 8.2.

Example: `aq21 input.aq21 -r`	Parameters and events are loaded from *input.aq21* file. Even if the *input.aq21* file contains definition of random seed, it is overwritten by automatic one.

In general AQ21 allows combining different parameters described above with some plausible constraints (eg. ***–F*** and ***–f*** parameters cannot be used together).

Example: `aq21 -F input -events -o output.aq21`	Parameters are loaded from *input.aq21* file. Events are loaded from input.data file and input.test file (if exists). Loaded events are displayed to the output whose name is output.aq21

4 INPUT FILES

4.1 AQ21 file format

The format of the AQ21 input file allows user to define all program parameters in a convenient way. The input file is a text file contains components for defining common domains of attributes, attributes, their types and domains (that may be common domains or domains defined together with attributes), and parameters of the planned runs of the algorithms for learning and testing.

The only program parameters that have to be defined in the input file are the decision classes to be learned and a list of attributes that correspond to the columns in the data tables (lists of events) provided to the program. All other parameters have default values, so the user does not have to specify them. These parameters can, however, be set differently to adjust the way program runs appropriately for a given problem.

An example of a simple AQ21 input file is presented below. A more complex example of an input file is presented in Appendix A. Information about AQ21 parser and list of reserved keywords is presented in APPENDIX D.

Problem_description *{* *This is an example of an AQ21 input file.* *}*	In the optional description component you can put any comments about the run, and it will be copied to the output.
Common_domains *{* *# The domains component is used to define domains that* *# can be shared among attributes* shape nominal {r, s, t} color nominal {r, y, b, g, w} *}*	In common domains user assigns types and lists of values with common domains that can be used later to define attributes.
Attributes *{* *# The attributes component is used to define attributes* *# for events in the data.* *x1 shape* *x2 shape* *x3 nominal {r, s, t, q, o}* *x4 color* *x5 color* *class nominal {1, 2, 3}* *}*	Each defined attribute corresponds to one column in input data table. Attributes can be either defined using common domains (eg. x1, x2, x4, x5) or their types and domains can be defined explicitly in this component (eg. x3, class).
Runs *{* *# The runs component is used to define parameters for* *# learning.* *Split_events_percentage = 0.6* *example_run_of_aq21* *{* *# Define specific parameters for this run* *Mode = TF*	Runs component is used to define classes and learning parameters. In this example one global parameter is used. Name of a run is any unique word. Parameters defined within a run affect only the run.

`  Consequent = [class = *]` `  }` `  another_example_run` `  {` `  Mode = TF` `  Ignore_attributes = class` `  Consequent = [x4 = r]` `  }` `}` **`Tests`** `{` `  # The tests component us used to define parameters for` `  # testing.` `  test1` `  {` `  Method = ATEST` `  Tolerance = 0.1` `  }` `}` **`Events`** `{` `  # In this component are the examples used for learning` `  # and testing.` `  s, s, r, r, r, 1` `  s, s, s, r, r, 1` `  s, s, t, g, r, 1` `  s, s, s, r, b, 1` `  r, t, s, r, r, 2` `  r, r, t, r, g, 2` `  r, s, r, r, g, 2` `  r, r, s, r, g, 2` `  r, s, q, r, g, 2` `  r, r, s, r, g, 2` `}`	*Consequent* is the only mandatory parameter. It is used to define classes for learning and/or testing. When * is used in the consequent program learns for all values of the attribute, otherwise it learns only for specified values. Definition of testing parameters. In this example one test is defined. It uses *ATEST* method implemented in AQ21. *Tolerance* is used to classify an event to more that one class if their degree of match is similar. Data table with training and/or testing events (examples). Each event is a list of comma separated values of attributes in order defined in the ***Attributes*** component. All values are mandatory (for missing or special values please refer to Section 7. Events are used for learning and/or testing (in this case both, since no testing events are explicitly specified in ***Testing_events*** component).

The input file provides AQ21 with information about training and testing data, and the way to execute rule learning and testing operations. The program allows the user to perform many different experiments in a single execution using the same input training and testing data.

The AQ21 input file is organized into components that reflect the organization of the program and provide a clear way to specify parameters. This section describes briefly the components of the input files, details are presented in the following sections (names of the components are keywords). Some of the components are optional and some are mandatory.

1. ***Problem_description*** (Optional)

   The component consists of comments about learning and testing problem, input files etc. It is ignored by AQ21, but is copied to the program output. It used to write any information that is desired in output (like the problem description).

2. ***Common_domains*** (Optional)

   Optional component that defines types and domains for which attributes may be defined (domains can also be defined in the ***Attributes*** component). For details please refer to Section 5 of this manual.

3. ***Attributes*** (Mandatory)

   Mandatory component that defines the attributes used in the input data. Each definition in the component corresponds to one column in the input data, in the order they are presented (columns are counted form left to right). For details please refer to Section 6 of this manual.

4. ***Runs*** (Mandatory)

   Mandatory component that defines classes and learning operations on the data. It consists of global parameters, and subcomponents that represent single runs of the program (at least one *run* subcomponent needs to be defined). Even if only testing is to be performed, this component needs to be specified. For details please refer to Section 8 of this manual.

5. ***Tests*** (Optional)

   Optional component that defines the testing of learned or input (predefined) hypotheses. The user can use testing algorithms such as ATEST and EPIC and their modifications. For details please refer to Section 9 of this manual.

6. ***Input_hypotheses*** (Optional)

   Optional component that is used to define input hypotheses for testing. The input hypotheses can be result of previous execution of AQ21 or can be manually written/modified by expert. One ***Input_hypotheses*** component corresponds to one class, so for example to define input hypotheses for three classes, three components are needed in input file. For details please refer to Section 9.4 of this manual.

7. ***PBM_input*** (Optional)

   Optional component that is used to manually define models for Prediction-based model testing (using EPIC-P). For details please refer to Section 9.4.3 of this manual.

8. ***Events*** (Optional)

   Optional component used to specify training events (examples) for the program. While this component is optional, events must be specified either using this component, or in a separate file as specified from the command line.

9. ***Testing_events*** (Optional)

   Optional component that specifies events (examples) used by the testing module of AQ21. The testing events can be also selected from training events or provided in a separate file.

An important feature of AQ21 is that its output can be used as input to a subsequent execution of the program. This feature makes experiments using AQ21 fully repeatable.

The user can also refine output hypotheses using a text editor, and apply them to the testing events again without additional work.

The user can make comments in the AQ21 input files. To comment out a single line put "#" in front of the line. To comment out a whole continuous part of a file (more or less than one full line), put "(#" at the beginning of the region to be commented, and "#)" at the end.

There are two main types of input files recognized by AQ21. The first type is in the format of the AQ21 input file described above that contains a definition of attributes, runs, tests, events etc. The second type of file is an input data file only that contains data to be used for training or testing. The data files are files with events (examples) in the same format as in the ***Events*** and ***Testing_events*** components. Each example is a list of comma separated values of attributes in order defined in the ***Attributes*** component. There is no comma at the end of example.

AQ21 is not case sensitive in terms of component names and parameters. For example it is correct to write an input file: Runs, runs, RUNS, ruNs, etc. Program is case sensitive for values of attributes as defined in ***Common_domains*** and/or ***Attributes*** components.

4.2 C4.5 file format compatibility

AQ21 has a feature that allows the user to load input files prepared for the C4.5 program. The C4.5 input is specified using three files having ".names", ".data", and ".test" filename extensions that define respectively attributes, training data, and testing data. They can be loaded using the *-f* option in the AQ21 command line followed by the filename stem common to both .names and .data files – without the extensions.

`aq21 -f robots`	AQ21 loads definitions of attributes from *robots.names* file, training data from robots.data file, and testing data from robots.test file (if exists). The files are in c4.5 format.

Although AQ21 is able to read c4.5 input files, it is not possible to specify any AQ21 parameters within this file format. Therefore, it is strongly recommended to modify ".names" file in the c4.5 format into ".aq21" file in the AQ21 format using any text editor (eg. notepad in Windows OS, or emacs in UNIX OS).

Example of simple ".names" c4.5 file and corresponding AQ21 file are presented below. Please note that in the AQ21 version of file more meaning is added to some attributes (for example type of attribute *holding* is structured, what is not available in c4.5).

`friendly, unfriendly.` `head: round, square, triangle.` `body: round, square, triangle.` `smile: no, yes.`	Example of c4.5 .names file. The first row specifies classes (last

.

`holding: sword, balloon, flag, us_flag, polish_flag.` `height:  short, medium, tall.` `antenna: red, yellow, blue, green, black, white.` `jacket:  red, yellow, blue, green, black, white.` `tie:     no, yes.`	column in data) and other rows specify attributes and their domains.

Common_domains `{` `  shape  nominal  {round, square, triangle}` `  color  nominal  {red, yellow, blue, green, black,` `  white}` `}`	AQ21 format. In order not to repeat definition of color and shape two common domains are defined.
Attributes `{` `  head  shape` `  body  shape` `  smile  nominal  {no, yes}` `  holding  structured {sword, balloon, flag, us_flag,` `  polish_flag}` `  {polish_flag --> flag` `  us_flag --> flag}` `  height  linear  {short, medium, tall}` `  antenna  color` `  jacket  color` `  tie  nominal  {no, yes}` `  class  nominal  {friendly, unfriendly}` `}`	Attributes as they appear in data. Types and domains of attributes head, body, antenna, and jacket are defined using common domains. Attribute holding is a **structured attribute** that is not available in c4.5. Class is defined as the last attribute as it is in the original data.
Runs `{` `  robots` `  {` `  Consequent = [class = *]` `  }` `}`	One simple run is defined in which only Consequent parameter is specified. Used '*' indicates learning for all values of attribute class (in this case *friendly* and *unfriendly*).
Tests `{` `  robots_test` `  {` `  Method = ATEST` `  }` `}`	One test is defined. It used ATEST classification program for classification of testing examples.

Data files in c4.5 format do not need be converted into AQ21 format. The only difference between the two formats in "." at the end of each event in c4.5 file and AQ21 is able to automatically remove it.

5 DEFINING COMMON DOMAINS OF ATTRIBUTES IN AQ21

5.1 Overview

All attributes that are defined in AQ21 need to be associated with a defined domain. There are two possible ways to define a domain for an attribute: (1) in the ***Common_domains*** component, and (2) explicitly, when the attribute is defined (see the ***Attributes*** component description). Although both methods are allowed, it is often preferable to define the domain first and later bind attributes to it, especially when more than one attribute uses the same domain. In addition, *counting attributes* (Section 8.2) require that the attributes to be counted explicitly have the same domain.

AQ21 supports most of the attribute types defined in Attributional Calculus (Michalski, 2004), namely, *nominal*, *linear*, and *structured*. The linear attributes are divided into *continuous*, *discretized continuous*, and *integer*. The type of an attribute depends on the structure of its domain. The following section explains how to define the attribute types and domains.

The structure of the ***Common_domains*** component in the AQ21 input file is presented below:

```
Common_domains
{
  first domain definition
  second domain definition
  …
  n-th domain definition
}
```

Example:

```
Common_domains
{
  color nominal {red, green, blue, yellow, black}
  height linear {short, medium, tall}
  length continuous 0.0 127.45
  seconds linear 60
}
```

5.2 Definition of Domains

The general form of a domain definition is:

domain name domain type [parameters]

where *domain name* is a unique name of the domain defined by the user, *domain type* is one of the types supported by AQ21, and *parameters* is an optional specification of additional properties of the defined domains (e.g., ranges, lists of values, etc.). The

following is the detailed description of the domain types: *nominal*, *linear*, *structured*, *continuous*, *integer*, and *discretized continuous*.

Nominal

Syntax:

domain_name nominal { list of values }

or

domain_name nominal size

Components:

domain name – a single word that specifies a unique name for the domain

list of values – a comma separated unordered list of values of the domain to be defined

size – a positive integer that specifies the number of values in the domain

Example:

```
color nominal {red, green, blue}
host nominal 22
```

Comment:

Nominal domains represent discrete, unordered sets of possible values. In AQ21, there are two ways of specifying such domains: (1) by listing all possible values, and (2) by defining the number of possible values. In the first case, the user must specify a comma-separated list of legal values enclosed in braces. In the second case, the user specifies an integer denoting number of values in the domain. In this case, the program will automatically generate values that are positive numbers 0, 2, 3, …, n-1, where n is the number specified by user. In the following examples, both definitions are equivalent:

Example 1:

```
host nominal 7
```

Example 2:

```
host nominal {0, 1, 2, 3, 4, 5, 6}
```

The minimum number of values of a nominal domain is two.

.

Linear

Syntax:

domain_name linear { list of values }

or

domain_name linear size

Components:

domain name – a single word that specifies a unique name for the domain

list of values – a comma separated, ordered list of values of the domain to be defined

size – a positive integer that specifies the number of values in the domain

```
Example:

    size linear {small, medium, large}
    host linear 22
```

Comment:

Linear domains represent discrete, ordered sets of values. In contrast to nominal attributes, the order of values is important, as there is a "greater-than" relationship defined between values. For example, the definition:

D linear {v1, v2, v3, v4, v5}

means that v1, v2, v3, and v4 are possible values of domain D, and v1 < v2 < v3 < v4 < v5, and that conditions such as [D <= v3] or [D = v2..v4] have meaning.

As can nominal domains, linear domains can be specified either by listing all possible values or by giving an integer number that denotes the number of possible values.

The minimum number of values of a linear domain is two.

Structured

Syntax:

domain_name structured { list of values } { list of relationships }

Components:

domain name – a single word that specifies a unique name for the domain

list of values – a comma separated list of values of the domain to be defined

list of relationships – a list that defines the relationships between the specified values, using --> operator

```
Example:

    place structured { World, Europe, Poland, Germany,
                       America, Mexico, US, Virginia,
                       Maryland, Alaska, Mars
                     }
    { Europe, America --> World
      Poland --> Europe
      Germany --> Europe
      US --> America
      Virginia --> US
      Maryland --> US
      Alaska --> US
      Mexico --> America
    }
```

Comment:

Structured domains represent partially ordered sets or hierarchies of possible values in which the partial order is defined by the --> relation where the "pointed" element is parent. Structured attributes can be used to represent hierarchies such as shapes, animals, or geographical data. Note that the list of relationships is not comma-separated.

The minimum number of values of a structured domain is two.

Continuous

Syntax:

domain_name continuous lower bound, upper bound

or

domain_name continuous

Components:

domain name – a single word that specifies a unique name of the domain

lower and upper bound – optional parameters that define the range of the continuous domain

```
Example:

    income continuous -1000.0, 3486.33
```

Comment:

The use of continuous domains allows a user to represent real values in both learning and testing. AQ21 can deal with continuous attributes without discretization that is required by most machine learning programs (performed either by user or automatically). For details on how generalization is applied to continuous attributes, please refer to the *epsilon* parameter described in Section 6.2.

Internally, continuous domains in AQ21 are implemented using *double* variables in C++, and all the limitations that apply to this type of variables apply to continuous domains and attributes.

Integer

Syntax:

domain_name integer lower bound , upper bound

or

domain_name integer

Components:

domain name – a single word that specifies a unique name for the domain

lower and upper bound – optional parameters that define the range of the integer domain

Example:

```
years integer -100, 100
```

Comment:

The integer domains were introduced to the program in order to allow user to deal with large linear domains. When the number of values of a linear domain is relatively big, it affects the performance of the program, which has to remember explicitly all values of the domain. When integer domains are used, the program stores only ranges of values (similarly to how it handles continuous domains).

Discretized Continuous

Syntax:

domain_name discretized continuous size , lower bound, upper bound

or

domain_name discretized continuous [list of points]

or

domain_name discretized continuous ranges [list of points]

Components:

domain name – a single word that specifies a unique name for the domain

lower and upper bound – parameters that define the range of the continuous domain

size – an integer value that defines the number of discretized points

list of points – a comma separated list of real values

```
Example:

    distance discretized continuous 100, 0.0, 4000.0

    temperature discretized continuous [ -50, -40, -30, -
20, -10, -5, 0, 0.5, 1, 2, 3, 4, 5, 10, 20, 50, 100, 200 ]

    length discretized_continuous ranges [0, 0.5, 1, 2, 3]
```

Comment:

As shown in the above example, the discretized continuous domains can be discretized either automatically, or manually by specifying a list of discretized points. In the first case, the program generates discretized equal-size intervals whose number is specified by the user (the *size* parameter in definition of the domain "distance" in the example above). It is also possible to invoke the ChiMerge algorithm (Kerber, 1992) to discretize a single attribute (Section 6.2).

Manual discretization can be done by specifying a list of discretization points explicitly. The smallest and the largest values in the list will be used respectively as the lower and upper bound for the domain.

Suppose that there are *n* discretization points x_1, x_2, …, x_n and $x_1 < x_2 < \ldots < x_n$. In such setting x_1 will be used as the lower bound of the domain, and x_n will be used as the upper bound of the domain. Real values in the range x_1 to $(x_1 + x_2)/2$ will be discretized as x_1. Real values in the range from $(x_{n-1} + x_n)/2$ to x_n will be discretized as x_n. For all x_i where $i \neq 1$ and $i \neq n$, real numbers in the range from $(x_{i-1} + x_i)/2$ to $(x_i + x_{i+1})/2$ will be discretized as x_i.

When the keyword *ranges* is used in the domain definition, the list of points is treated as a list of borders between intervals. In the above example, in the definition of the domain "length," the intervals are: 0 to 0.5, 0.5 to 1, 1 to 2, and so on.

6 DEFINING ATTRIBUTES IN AQ21

6.1 Overview

The previous section described how to define domains that can be used to define attributes. However, as mentioned before, this is not the only way to define attributes. They can be defined either by specifying domains as defined before, or by explicitly specifying the domain type. The details on how to do it follow.

For a description of types of attributes that are available in AQ21 please refer to Section 5.

The attributes in AQ21 are defined in an ***Attributes*** component that has the following form:

```
Attributes
{
  first attribute definition
  second attribute definition
  …
  n-th attribute definition
}
```

Each row in the attributes definition corresponds to exactly one column in the input data (top row corresponds to the leftmost column in the data). Every column in the input data can be defined using a standard attribute or a meta-attribute, or can be ignored.

Example:

```
Attributes
{
    x1              height    cost = 2
    body            nominal    { square, round, triangle }
    ignore
    location        place
    episode
    significance
}
```

In the above example the attribute x1 is defined based on a common domain height (defined previously), and with cost equal to two. Body is a nominal attribute with three possible values. The third column in the input data is to be ignored. Location is defined using the place domain that was defined in the ***Common_domains*** component. Episode and Significance are two meta-attributes that are described in Section 6.3.

6.2 Attribute Definition

To define an attribute, the following line of text is used:

attribute name attribute domain attribute options

Attribute name is a unique name of the attribute. The name can be used later in all parts of the AQ21 input file and, for example, in the learned rules.

Attribute domain is either the name of a domain already defined or a definition of the domain. For details on domain definition, see Section 5.

Attribute options are used to define additional features of the attributes. The possible options are *cost*, *epsilon*, and *discretization*.

The *cost* of an attribute is a positive integer, whose value can be interpreted as the relative expense of evaluating the attribute. It is used by the *mincost* LEF criterion (see Section 8.3) to minimize the cost of the learned rules (e.g., Michalski and Kaufman, 2001).

Epsilon provides the program with information on how to generalize continuous attributes. Its value can be between zero and one where zero means maximum generalization (a condition is maximally general so as not to cover a negative example) and one means minimum generalization (a condition is maximally specific without losing any positive coverage).

Discretization followed by the desired number of ranges is introduced ONLY for continuous attributes and invokes an automatic program that discretizes the data. Currently the ChiMerge discretization method (Kerber, 1992) is implemented. In the example below attribute "width" will be discretized into seven intervals using the ChiMerge method.

Example:

```
body    nominal     { square, round, triangle }  cost = 1
length  continuous  cost = 3   epsilon = 0.2
width   continuous  ChiMerge 7
```

6.3 Meta-attributes

There are four predefined, special attributes, called *meta-attributes*, that express information about the events, namely: *key*, *episode*, *frequency*, and *significance*. The most significant difference between attributes and meta-attributes is that meta-attributes are not used directly in the learning process (they are never components of learned hypotheses); they can only be used to control the AQ21 learning and testing as described below.

Key is a unique positive integer that is used to identify the event.

.

Episode is used to distinguish between series or sequences of events. For example one series of activities of a computer user during one session can be viewed as one episode. The episode parameter does not affect the learning process; however it is used during testing when one of EPIC methods is applied and the program classifies entire episodes instead of single events.

Frequency is a positive integer value that indicates how many times an event appears in the data. The events that appear more than once in the data can have their frequency fields set to values greater than one, they can appear in the data more than once, or both. The value of frequency is important for the program execution, and is used, for example, by many of the LEF criteria.

Significance is a real number that indicates how important the event is. It is used by the *maxsignificance* LEF criterion to give preference to rules that cover the most important events (see the LEF criteria descriptions, Section 8.3).

7 META-VALUES IN AQ21

In several situations, the exact value of an attribute is not provided to the program. Such situations may happen, for example, when the value is not known, or the user decided not to use it for learning. AQ21 implements three meta-values that can be used instead of real values in the attribute domains. A theoretical description of the meta-values can be found in (Michalski, 2004; Michalski and Wojtusiak, 2005). The three meta-values implemented in AQ21 are described below.

The "*Missing*" (a.k.a. *don't know* and *unknown*) meta-value, denoted in data by "?", represents situations in which the value of an attribute exists, but for some reason is not known. For example, it may not have been measured, or for other reason it is not present in the data.

The "*Irrelevant*" (also known as "*Don't care*") meta-value, denoted in data by "*", represents situations in which a value is irrelevant for learning the specific class. For example color of the eyes is rather irrelevant when determining a student's grade. Information that an attribute is irrelevant for a given event is entered into data by a domain expert.

The "*Not applicable*" meta-value, denoted in data by "NA" or "N/A", represents situations in which the value of an attribute makes no sense for a given event. For example color of a jacket is not applicable when there is no jacket.

The example below illustrates a toy problem in which meta-values are used (in this case "N/A" and "?" are used). It shows only the definition of attributes and events (no other components of the AQ21 input are specified).

Example:

```
Attributes
{
  head_shape nominal { square, round, triangle}
  body_shape { square, round, triangle }
  wearing_jacket { yes, no }
  color_of_jacjet { red, green blue }
}

Events
{
  square, ?, yes, red
  round, triangle, no, N/A
  round, triangle, yes, blue
}
```

.

8 DEFINING LEARNING PARAMETERS IN AQ21

8.1 Overview

The learning parameters are organized in the ***Runs*** component of the AQ21 input file. The structure of this component is presented below:

```
Runs
{
  global parameters

  run_one
  {
    parameters for the first run
  }

  run_two
  {
    parameters for the second run
  }
  …

  run_n
  {
   parameters for the n-th run
  }
}
```

Example:

```
Runs
{
  Output = group

  Split_events_percentage = .7

  good
  {
    Consequent = [ group = good ]
    Maxstar = 1
    Maxrule = 1
    Trim = Optimal
    Mode = PD
    Display_selectors_coverage = true

    LEF_sort
    {
      MaxPositives, 0
    }


    LEF_ps
    {
      MaxQ, 0
```

```
          MinComplexity, 0
        }
      }

      bad
      {
        Consequent = [ group = bad ]
        Trim = Optimal
        Mode = PB

    }
```

The ***Runs*** component contains definitions of global parameters that apply to all runs, followed by definitions of specific runs. Inside the runs component, at least one run must be defined, and each run must have a unique name (as in the example above).

The definition of a parameter's values takes two possible forms:

parameter = value

or

parameter = list of comma separated values.

```
Example:

    Consequent = [ group = good ]

    Ignore_attributes = color, height
```

8.2 Global Learning Parameters

Global parameters are common to all defined runs and apply to all training and testing events. Some of them, however, are identical to run-specific parameters, and can be defined either as global parameters that apply to all runs or in run-specific components that apply to a single run. **All Global Parameters are optional; none are mandatory.** A list of the global parameters defined in AQ21 is presented below in alphabetical order.

Attribute selection method

Syntax:

Attribute_selection_method = method

Possible values:

promise, gain_ratio, none

Default value:

none

```
Example:

    Attribute_selection_method = promise
```

Comment:

In large datasets that have many attributes it is important to select the most relevant attributes before starting the learning process. In many applications, attempting the learning process on the full dataset with all attributes is too slow and inefficient. Here, the selected method is used to determine the attributes that have the highest chance to be relevant for the learning. Two criteria: *promise* (Baim, 1982; Kaufman, 1997) or *gain ratio* (Quinlan, 1993) can be used to determine which attributes are likely to be the most relevant.

In AQ21 to select attributes for learning we compute the *discriminatory power* (*dis-power*) of each attribute and compare it with the acceptance threshold. Attributes whose dis-power is below the threshold will not be used for learning. The complete algorithm can be written in three steps shown below.

```
1. Given a dataset, evaluate the discriminatory power of
   each individual attribute according to some measure
   (applied to each class vs. the rest of the classes)
2. establish an acceptance threshold for attributes
   depending on the number of attributes in the data
3. remove from dataset the attributes whose dis-power is
   below the threshold
```

To make the method more general, it is permissible that for each decision class, the set of attributes that will be used for learning can be different. Note that the discriminatory power in step 1 is computed for each class against the collective rest of the classes.

AQ21 gives three possibilities of selecting the acceptance threshold parameter. First, it can be defined explicitly by the user. For instance, an acceptance threshold set to 0.5 will filter out all attributes whose dis-power is below 0.5. The user can also define the parameter *attribute selection tolerance*, which specifies the acceptance threshold using the following formula (MIN and MAX are respectively minimum and maximum values of dis-power for evaluated attributes):

*acceptance threshold = MIN + (MAX – MIN) * acceptance threshold tolerance.*

If in the AQ21 input file the user defines both the acceptance threshold and the acceptance threshold percentage, the acceptance threshold is ignored and computed as described above.

There are two implemented methods for determining the dis-power of attributes – *Promise* and *Gain Ratio*. Both return values in the range [0, 1] that can be easily compared with the acceptance threshold.

Promise

The implemented method is a simplified *Promise* (Baim, 1982; Kaufman 1997) method that uses two classes represented by positive and negative events. The negative events are all events that are not from the class that is being considered. The algorithm for attribute x and class C can be described by the following pseudo-code:

```
P = 0
For each value of the attribute v_i
    S = { examples : x = v_i }
    Find class C that has the largest number of examples in S
    P = P + #C∩S/#S
Return P
```

After a simple normalization the returned value is in the range [0, 1].

Gain Ratio

The Gain Ratio is a well known attribute quality measure used by programs such as C4.5 for building decision trees (Quinlan, 1993). The Gain Ratio is defined as

$$gain\ ratio(X) = gain(X)/split\ info(X)$$

where split info represents the potential information gained by splitting examples by using attribute X, and gain is the information that is gained by partitioning events by using attribute X. For exact formulas please refer to for example (Quinlan, 1993).

Attribute selection threshold

Syntax:

Attribute_selection_threshold = value

Possible values:

real number in the range from 0 to 1 inclusive

Default value:

0

Example:

```
Attribute_selection_threshold = 0.5
```

Comment:

This parameter is used to define the minimum value of an attribute's discriminatory power for the attribute to be used for learning. For details, see the description of the *attribute selection method* parameter. Note that this parameter is ignored whenever the parameter *attribute selection tolerance* is defined.

.

Attribute selection tolerance

Syntax:

Attribute_selection_tolerance = value

Possible values:

real number in the range 0 to 1

Default value:

0

```
Example:
    Attribute_selection_tolerance = 0.5
```

Comment:

This parameter is used to define a tolerance for the minimum value of an attribute's discriminatory power in order to determine if the attribute is to be used for learning. If this parameter is present, the acceptance threshold is computed using the following formula:

*attribute selection threshold = MAX - (MAX – MIN) * attribute selection tolerance*,

where MIN and MAX are the minimum and maximum attribute discriminatory powers found, respectively. For further details, please refer to *attribute selection method* parameter description.

Counting Attribute

Syntax:

Counting_attribute = value

Possible values:

true, false

Default value:

false

```
Example:
    Counting_attribute = true
```

Comment:

This option is used to enable the *Counting Attributes* generation in AQ21. The counting attributes are automatically constructed for all input nominal or linear attributes with the same domains. For instance:

count(x1, x2, x3 = r) = 2

means that value of exactly two attributes of the three (x1, x2, x3) are equal to r.

Output and ignored attributes must be defined in this component before definition of the counting attributes (please refer to *Output_attributes* and *Ignore_attributes* parameters). Thus, output attributes are the same for all runs defined in the run-specific components.

Cross Validation

Syntax:

Cross_validation = k

Possible values:

integer

Default value:

0

Example:

```
Cross_validation = 3
```

Comment:

Cross Validation runs AQ learning and testing a number of times that is specified by parameter *k*, on different subsets of data. The program splits the input data into *k* subsets D1, D2, …, Dk and applies learning and testing in the following way:

1. Learn on D1, …, D(k-1) and test on Dk
2. Learn on D1, …, D(k-2), Dk and test on D(k-1)

…

k. Learn on D2, …, Dk and test on D1

A summary of the cross validation operation is displayed in terms of the average of the primary learning and testing results (e.g., accuracy, precision, number of rules).

Events Percentage

Syntax:

Events_percentage = ratio

Possible values:

real value between 0 and 1

Default value:

1.0

Example:

```
Events_percentage = 0.9
```

Comment:

The *Events percentage* parameter is used to control the percentage of input events that are used for training. If, for instance, the parameter is set to 0.9, the program will use 90% of the training events, and the other 10% will be ignored.

Ignore Attributes

Syntax:

Ignore_attributes = attribute1 [,attribute2 [, …]]

Possible values:

comma-separated list of attributes

Example:

```
Ignore_attributes = x1, x3, color, height
```

Comment:

Parameter used to ignore attributes from original data for the learning process. The selected attributes are ignored for learning in all runs defined in the ***Runs*** component.

The *Ignore_attributes* parameter can also be used as a run-specific parameter to ignore attributes only in the run in which it is specified. If ignore attributes are defined in both places (global and run-specific parameters), the union of the attributes is ignored in a given run.

Output attributes

Syntax:

Output_attributes = attribute1 [, attribute2 [, …]]

Possible values:

comma-separated list of attributes

Example:

```
Output_attributes = color
```

Comment:

This option is used to specify the output attributes used in the learning and testing phases. They must be consistent with consequents defined in run-specific components. This option is mandatory when any of the constructive induction methods are used (for example counting attributes).

Random Seed

Syntax:

Random_seed = seed

Possible values:

integer

Example:

```
Random_seed = 123
```

Comment:

This number specifies an initial value for the random number generator used in the program. There are several places in the algorithm where values are selected randomly, and in order to be sure that results are precisely repeatable when desired, there must be such option. If not selected, its value is generated using current date and time.

Split Events Percentage

Syntax:

Split_events_percentage = ratio

Possible values:

real value between 0 and 1

Default value:

1.0

Example:

```
Split_events_percentage = 0.8
```

Comment:

The Split Events Percentage is used to set the ratio between number of training and testing events. For instance, a value 0.8 means that 80% of the events will be used for training, and the other 20% will be used for testing. Selection of events is done automatically by the program.

Testing Events Percentage

Syntax:

Testing_events_precentage = ratio

Possible values:

real value between 0 and 1

Default value:

1.0

Example:

```
Testing_events_percentage = 0.9
```

Comment:

The *Testing Events Percentage* parameter is used to control the fraction of events that are used for testing. If, for instance, the parameter is set to 0.9, the program will use 90% of the testing events, and 10% will be ignored.

8.3 Run-Specific Learning Parameters

Run-specific parameters are those that are defined for a particular run only. They do not affect other runs. The reason for this is that a user may want to use a specific set of parameters when learning for one class and another set of parameters when learning another class. Run-specific parameters are defined inside the ***Runs*** component after the global parameters by specifying the unique run's name followed by "{". The end of the definition of run-specific parameters for a single run is specified by the symbol "}". A user can define as many runs as is needed.

The only parameter in run-specific component that has to be specified is *Consequent* (in learning mode only – see *Mode* parameter). All other parameters are optional, and their values are taken either from the global parameter settings, or program defaults.

```
Example:
Runs
{

    my_run
      {
        Consequent = [ group = * ]
        Maxstar = 1
        Trim = Optimal
        Mode = TF
      }
…
}
```

Additional stars

Syntax:

Additional_stars = value

Possible values:

Integer

Default value:

0

```
Example:

    Additional_stars = 5
```

Comment:

In the Theory Formation mode (see below), AQ21 learns complete and consistent rulesets. This also means that covering all positive events is a termination condition for the learning process. In some cases, it may be desirable to generate additional stars even if the learned cover is already complete, and this parameter specifies how many times to do so.

Ambiguity

Syntax:

ambiguity = value

Possible values:

IncludeInPos, *IncludeInNeg*, *IgnoreForLearning*, *IncludeInMajority*, *DetectAmbiguity*

Default value:

IncludeInPos

Example:

```
Ambiguity = IgnoreForLearning
```

Comment:

This parameter is used to control how AQ21 handles ambiguous data (identical events in both the positive and negative classes). Five possible methods are described below.

IncludeInPos	Ambiguous events are used as positive examples for learning.
IncludeInNeg	Ambiguous events are used as negative examples for learning.
IgnoreForLearning	Ambiguous events are removed (ignored) from the training data.
IncludeInMajority	Ambiguous events belong to the class for which they occur the most often (their frequency is the highest).
DetectAmbiguity	The program checks if there are ambiguous events in the training data. If so, a list of the ambiguous events is displayed, otherwise learning is performed as normal.

Attribute selection method

Syntax:

Attribute_selection_method = method

Possible values:

promise, gain_ratio, none

Default value:

none

Example:

```
Attribute_selection_method = promise
```

Comment:

Please refer to the description in global parameters in Section 8.2.

.

Attribute selection threshold

Syntax:

Attribute_selectoin_threshold = value

Possible values:

real number in range from 0 to 1

Default value:

0

```
Example:
    Attribute_selection_threshold = 0.5
```

Comment:

Please refer to the description in Section 8.2.

Attribute selection tolerance

Syntax:

Attribute_selection_tolerance = value

Possible values:

real number in range from 0 to 1

Default value:

0

```
Example:
    Attribute_selection_tolerance = 0.5
```

Comment:

Please refer to description in Section 8.2.

Compute Alternative Covers

Syntax:

Compute_alternative_covers = value

Possible values:

true, false

Default value:

false

```
Example:
    Compute_alternative_covers = true
```

Comment:

The learning process can usually produce more than one alternative description of a learned class (alternative cover). AQ21 generates alternatives from final rules generated in the star generation process. This is done only if the *maxrule* parameter is greater than one and this parameter is set to *true*. When the option is enabled, program computes displays the alternative covers in the ***Output_hypotheses*** component. The maximum number of displayed alternatives is controlled by the *Max_alternatives* parameter.

Consequent (class definition)

Syntax:

Consequent = complex

Possible values:

complex

```
Example:
    Consequent = [ color = red, blue ]
```

Comment:

Consequent defines the class(es) for which learning is to be performed by AQ21. It defines all left sides of rules that describe learned classes. The value of the consequent is given by a complex (Michalski, 2004). The consequent is used to determine positive and negative examples for the runs (those who match consequent are positive examples).

If the parameter *Output_attributes* is used in the global parameters (Section 8.2), then all specified output attributes must be used in the consequent.

Note:

To learn rules that distinguish between all possible values of an attribute, use the complex in the form [attribute = *], where attribute distinguishes between classes. The possible

values of the attribute are taken from its domain definition, not from the actual data. Because of that, it may happen that there are classes without any positive examples.

Example:

```
consequent = [ color = * ]
```

Continuous Optimization Probe

Syntax:

Continuous_optimization_probe = value

Possible values:

positive integer

Default value:

5

Example:

```
Continuous_optimization_probe = 10
```

Comment:

Used for optimization of continuous selectors (conditions). This parameter defines the maximum number of possible extensions of an interval during the optimization of continuous selectors. This parameter works only when AQ21 is operating in *PD* or *ATF* modes, and the *Optimize_ruleset* parameter is on. For details on ruleset optimization please refer to the *Optimize_ ruleset* parameter description on page 51.

Counting Attribute

Syntax:

Counting_attribute = value

Possible values:

true, false, ga

Default value:

false

```
Example:

    Counting_attribute = true
```

Comment:

The *Counting_attribute* parameter specified in the run-specific component is similar to one that is defined in the global parameters, but works only locally within a specific run.

In addition to the values available in the global *Counting_attribute* parameter, the one specified in the run-specific component also takes the value "ga" (which stands for *Genetic Algorithm*). The option must be followed by four integer numbers that specify: population size, number of generations, number of children, and maximum number of attributes to be used for constructing the new attributes. For a description of the parameters' meanings, please refer to Evolutionary Computation literature for example (Michalewicz, 1994).

Define (create) new attribute

Syntax:

Define attribute_name [parameters] = definition

Possible values:

attribute_name – unique name of attribute to be constructed

definition – arithmetic expression that is used to define the attribute

parameters – definition of attribute parameters such as *epsilon* and *cost*

```
Example:

    Define x4 = x1 + x2 * sin( x3 )
    Define x5 [epsilon = 0.2, cost = 3] = x1 * x4
```

Comment:

There are situations where it is not possible to describe a desired concept using the original attributes. It may also happen that the user knows that there may be an arithmetic interrelation between existing attributes, and wants to represent the relationship in the program. The simplest way to do this is to define new attributes in form of *A-Rules* (*Arithmetic Rules*) that explicitly specify new attributes.

AQ21 allows the user to define A-Rules using only continuous and integer attributes. The generated attributes are continuous. The following operators can be used to define A-Rules: +, - (binary minus), *, /, ^ (power), - (unary minus), and functions: *max(ex1, ex2*

), min(ex1, ex2), avg(ex1, ex2), abs(ex), sin(ex), cos (ex), tan(ex). In the list of functions *ex1*, *ex2*, *ex* are valid expressions.

In cases where the value of the derived attribute does not exist (for example division by zero) AQ21 assigns the value *N/A* (*Not Applicable*, described in Section 7).

Display Events Covered

Syntax:

Display_events_covered = value

Possible values:

true, false

Default value:

false

Example:

```
Display_events_covered = true
```

Comment:

If this parameter is turned on, in the output hypotheses component for the current run, after each rule, the program displays a component named *covered_positives* that contains positive examples covered by the rule, and *covered_negatives* that contains negative examples covered by the rule.

Display Selectors Coverage

Syntax:

Display_selectors_coverage = value

Possible values:

true, false

Default value:

true

Example:

```
Display_selectors_coverage = false
```

Comment:

This option is used to indicate if the user wants to display coverage (numbers of positive and negative examples covered) of selectors when displaying rules. Although rules are displayed at the end of output, this affects only rules that are learned in the current run. For instance a rule displayed with and without coverage of selectors is presented below.

```
selectors_coverage = true:

<-- [x2=s : 4,0]
      : p=4,np=4,u=4,cx=7,c=1,s=4 #92

selectors_coverage = false:

<-- [x2=s]
      : p=4,np=4,u=4,cx=7,c=1,s=4 #92
```

Display Values Covered

Syntax:

Display_values_covered = value

Possible values:

true, false

Default value:

false

```
Example:

    Display_values_covered = true
```

Comment:

If this parameter is activated, when displaying rules, after each selector, AQ21 displays the coverage of each value of the attribute used in the selector. The example below shows an example rule with the list of values covered by selectors.

The example below shows rule with one condition that states *x6>=0.35*. The information written after the condition is a list of covered values that appear in the training dataset. For example value 0.5 is present in one positive and zero negative training examples, which is denoted by " :1, 0;", similarly value 1.4 is also present in one positive and zero negative examples. In the second line of the example, a rule summary is presented (the rule has only one condition). For a description of the parameters displayed with the rule please refer to Section 10.2.

.

Example:

```
<-- [x6>=0.35 : 3,0] (# 0.5 :1, 0; 1.4 :1, 0; 2 :1, 0;  #)
    : p=3,np=3,enp=3,n=0,en=0,u=3,cx=5,c=1,s=3 # 30
```

Exceptions

Syntax:

Exceptions = value

Possible values:

true, false

Default value:

false

Example:

```
    Exceptions = true
```

Comment:

In *Pattern Discovery* (PD) and *Approximate Theory Formation* (ATF) modes AQ21 can learn rules with exceptions (Michalski, 2004).

Ignore Attributes

Syntax:

ingnore_attributes = attribite1 [, attribute2 [, …]]

Possible values:

comma-separated list of attributes

Example:

```
    ingnore_attributes = color, height, age
```

Comment:

The parameter is identical to one that is defined in the global parameters component, but works only for a specific run. For more details please refer to the Section 8.2.

Ignore events

Syntax:

Ignore_events = rules

Possible values:

a set of rules

```
Example:

    Ignore_events <-- [color = red, yellow][shape = square]
                  <-- [color = blue]
```

Comment:

This option provides the possibility of ignoring events in the learning process. The user can specify a list of rules that describe events to be ignored. Strict matching is used when matching events with these rules.

Learn Rules Mode

Syntax:

Learn_rules_mode = value

Possible values:

standard, multi_seed

Default value:

standard

```
Example:

    Learn_rules_mode = multi_seed
```

Comment:

This parameter controls the selection of seeds used to generate stars. In *standard* mode, AQ21 randomly selects one seed for star generation. In *multi-seed* mode, the program selects randomly *n* seeds that are extended against the first negative event (the value of *n* is specified using the *number_of_seeds* parameter). In the next step, AQ21 selects the partial star with the highest value of *new positives Q* (please refer to the LEF criteria described below).

.

Lexicographical Evaluation Function (LEF)

Syntax:

```
lef_type
{
[(criterion1, tolerance1)]
[(criterion2, tolerance2)]
…
}
```

Possible values:

lef_type – *LEF_star*, *LEF_partial_star*, *LEF_sort*

criterion – *MaxNewPositives*, *MaxNewPositivesQ*, *MaxPositives*, *MinNegatives*, *MinNumberSelectors*, *MaxNumberSelectors*, *MinCost*, *MaxSignificance*, *MaxQ*, *MaxEstimatedPositives*, *MinEstimatedNegatives*, *MaxEstimatedQ*, *GainRatio*

tolerance – real value in range from 0 to 1

Default values:

TF and ATF modes:

```
lef_ps
{
      MaxNewPositives, 0
      MinComplexity, 0
      MinNumSelectors, 0
      MinCost, 0
}

lef_star
{
      MaxNewPositives, .3
      MinComplexity, .3
      MinNumSelectors, .3
      MinCost, .3
}

lef_sort
{
      MaxPositives, 0
      MinComplexity, 0
}
```

PD mode:

```
lef_ps
{
      MaxNewPositivesQ, 0
      MaxNewPositives, 0
      MinNumSelectors, 0
      MinComplexity, 0
      MinCost, 0
```

```
}

lef_star
{
      MaxNewPositives, .3
      MinComplexity, .3
      MinNumSelectors, .3
      MinCost, .0

lef_sort
{
      MaxQ, 0
      MaxPositives, 0
      MinComplexity, 0
}
```

Example:

```
      lef_ps
      {
            MaxNewPositives, 0.1
            MinNumSelectors, 0

      }
```

Comment:

Lexicographical Evaluation Function (LEF) provides a simple and efficient way of specifying a multi-criteria evaluation. The LEF is defined as a list of pairs <criterion, tolerance>. Elements are evaluated criterion after criterion in order from the list. All elements that are beyond the tolerance from the best value of a criterion are filtered out.

LEF is applied during a partial star generation (LEF-PS), to evaluate rules in generated stars (LEF-STAR), and to sort rules in the program output (LEF-SORT).

MaxPositives

Maximum Number of Positives can be applied when user wants AQ21 to prefer rules that cover the largest number of positive events. This criterion is time-consuming, since it has to count the number of positive examples that are covered.

MaxEstimatedPositives

Maximum Estimated Number of Positives covered by a rule. This criterion is based on an estimate (upper bound) of the number of positives covered by a rule.

MinNegatives

Minimum Number of Negative Examples can be applied when the user wants AQ21 to prefer rules that cover the smallest number of negative examples. This criterion is time-consuming since it has to count number of negative examples that are covered by the rule.

MinEstimatedNegatives

Minimum Estimated Number of Negatives covered by a rule. It is based on an estimate of the number of negatives and can be used to replace the MinNegatives criterion in order to increase program speed.

MaxNewPositives

Maximum Number of New Positives covered by a rule. Please note that in order to compute this value, the program needs to check all positives and check if they are already covered or not, which is very time consuming for large datasets.

MaxEstimatedNewPositives

Maximum Estimated Number of Positives covered by a rule. This criterion can be used instead of MaxPosivesNewPositives criterion, since it is faster. However its result is an estimate (upper bound) of the number of new positives that may not be correct.

MaxQ

Maximum Q is the most time-consuming LEF Criterion. It maximizes the value of Q(w) (see the "minimum q" parameter, page 48). It is very extensively used in the PD and ATF modes of AQ21 for selection of best rules.

MaxEstimatedQ

Max Estimated Q is used to maximize the value of the Q(w) measure based on estimation of positive and negative events covered by a rule. For definition of the Q(w) measure, please refer to the minimum q parameter.

MaxNewPositivesQ

Maximum New Positives Q is used to maximize Q(w) based on new positives. It is very time-consuming, since it has to compute the numbers of new positives covered by the rule and negatives covered by the rule. For a definition of Q(w) measure please refer to the description of the minimum q parameter.

MaxEstimatedNewPositivesQ

Maximum Estimated Number of Positives Q is used to maximize value of Q(w) based on an estimate of new positives and new negatives covered by a rule. It can be used to replace MaxNewPositivesQ criterion since it is faster. For a definition of Q(w) measure, please refer to the description of the minimum q parameter.

MaxNumSelectors

Maximum Number of Selectors can be applied when the user prefers rules that contain more attributes (a larger number of selectors). This measure should not be used as a first criterion for LEF_PS and LEF_STAR, because it does not include any information about positive and negative examples covered.

.

MinNumSelectors

Minimum Number of Selectors can be applied when the user prefers rules with smaller numbers of selectors.

MaxSignificance

The Maximum Significance LEF criterion is used to prefer rules with the highest significance. Significance of a rule is defined as the sum of the significances of the positive examples covered by the rule. Significance of an event is provided by user using the significance meta-attribute (see Section 6). To compute the value, it is necessary to loop through all positive examples, which is time-consuming for large datasets.

MinComplexity

Minimum Complexity is used to minimize complexity of learned rules. The complexity of a rule is measured as complexity of premise + 2 * complexity of exception. Complexity of a complex (either premise or exception) is measured as the sum of the complexities of its single selectors. Weights of operators in selectors are presented in the table below.

conjunction	4
disjunction	10
internal disjunction	2
range	2
less or grater	1
equal	1
Not equal	2

Table 2: Weights used to compute complexity

MinCost

The Minimize Cost LEF criterion is used to select rules with the lowest cost of attributes. As was described in Section 6.2, each attribute has an assigned cost. The cost of a rule is defined as the sum of the costs of the attributes included in the rule.

Maxrule

Syntax:

Maxrule = value

Possible values:

positive integer

Default value:

5

.

Example:

```
Maxrule = 2
```

Comment:

Maximum number of rules kept from each star (only one is required) in order to improve performance. *Maxrule* greater than one is also required when alternative covers are generated.

Maxstar

Syntax:

Maxstar = value

Possible values:

positive integer

Default value:

2

Example:

```
Maxstar = 5
```

Comment:

Maxstar parameter defines the maximum number of rules kept in memory during the star generation (Michalski and Kaufman, 2001). This parameter is used to narrow down the search space over all possible rules that can be learned using *beam search*. Selection of the best rules is done according to criteria defined by the *Lexicographical Evaluation Function* (LEF-PS).

Model

Syntax:

Model = complex

Possible values:

complex

Example:

```
Model = [user = user2][host = host13]
```

Comment:

This parameter is used to define a model in the *Prediction Based Model*. During the learning phase, AQ21 will ignore all events (both positive and negative) that do not match the model. The definition of model also affects the prediction-based testing (EPIC-P) defined for the current run (for details on EPIC parameters please refer to the Section 9.2)

Minimum u

Syntax:

Minimum_u = value

Alias:

Minimum_rule_coverage

Possible values:

integer

Default value:

1

```
Example:

    Minimum_u = 10
```

Comment:

This option defines a minimum *unique coverage* of a rule that is required for it to be added to a ruleset. This parameter works only in TF mode, and only when *truncate* parameter is set to *true*.

Minimum Q Percentage

Syntax:

Minimum_q_percentage = value

Possible values:

double

Default value:

.5

```
Example:

    Minimum_q_percentage = 0.2
```

Comment:

This parameter defines a minimum acceptable value of *Q(w)* for learned rules, where $Q(R, w) = compl(R)^{w} * consig(R)^{(1-w)}$. The minimum Q is computed as the best Q found multiplied by the minimum Q percentage. *Compl(R)* is a measure of completeness of the rule *R*, and *consig(R)* is a measure of consistency of the rule *R*. In *Pattern Discovery* mode, AQ21 removes rules whose value of Q(w) is below the minimum.

Mode

Syntax:

Mode = value

Possible values:

TF, PD, ATF, TEST

Default value:

TF

```
Example:

    Mode = PD
```

Comment:

Learning mode is used to select which algorithm will be used for rule learning. In TF (*Theory Formation*) mode, learned rules are complete and consistent, while in PD (*Pattern Discovery*) and ATF (*Approximate Theory Formation*) modes, they may be neither complete nor consistent. Rules learned in PD and ATF modes are optimized according to value of Q(w). In the PD mode, AQ21 optimizes rules while learning them (in the star generation phase), while in the ATF mode initially complete and consistent rules are learned (as in the TF mode), but later the rules are optimized according to their Q(w) measure, which may cause a loss of completeness and/or consistency.

In the *TEST* mode, the program does not learn rules, but only passes parameters to the testing module (please refer to the ***Tests*** component).

Negatives Percentage

Syntax:

Negatives_percentage = value

Possible values:

real value in range from 0 to 1

Default value:

0.8

```
Example:

    Negatives_percentage = 0.5
```

Comment:

This parameter controls the stop condition for the star generation algorithm. If there is no progress after extending against *negatives probe* times, AQ21 selects another seed (and starts generating a new star). The *negatives probe* is defined as *negatives percentage* multiplied by the number of negative examples.

Number of Seeds

Syntax:

Number_of_seeds = value

Possible values:

positive integer

Default value:

10

```
Example:

    Number_of_seeds = 3
```

Comment:

This parameter controls the number of seeds used in the multi-seed star generation algorithm. For details please refer to the *Learn Rules Mode* parameter description on page 42.

.

Optimize ruleset

Syntax:

Optimize_ruleset = value

Possible values:

true, false

Default value:

true

Example:

```
Optimize_ruleset = false
```

Comment:

This option affects learning in the ATF and PD modes. If set to true, the program optimizes learned rulesets (1) after each star is generated, and (2) the final rulesets learned by program. As an effect of the optimization, learned rulesets may be neither complete nor consistent. Optimization of rules may involve dropping of selectors, extension of intervals, and climbing the hierarchy trees for structured attributes.

Trim

Syntax:

Trim = value

Possible values:

MostGen, MostSpec, Optimal

Default value:

Optimal

Example:

```
Trim = MostGen
```

Comment:

A parameter used to control the generality of rules. The trimming algorithm is applied to learned rules in order to specialize them (note that learned rules before trimming are maximally general descriptions of the learned concept). In *MostGen* mode, learned rules remain unchanged. In *MostSpec* mode, they are specialized to cover only these values

that appear in positive examples, and the learned rules contain all attributes (the most general rule is intersected with the refunion of positive examples). In optimal mode, the rules also cover only values from positive examples, but use only attributes that are required to distinguish positive from negative.

Truncate

Syntax:

Truncate = value

Possible values:

true, false

Default value:

true

Example:

```
Truncate = false
```

Comment:

An option used to determine if rules should be truncated in TF mode. If set to true, AQ21 removes rules with unique coverage levels lower than *minimum u* parameter. The truncation is applied after all rules are learned.

w

Syntax:

w = value

Possible values:

real number in range from 0 to 1

Default value:

0.5

Example:

```
w = 0.1
```

Comment:

W is a parameter that controls the tradeoff between consistency and completeness of learned rules in *Pattern Discovery* and *Approximate Theory Formation* mode. It is used in the *Q(w)* measure. For details of the *Q(w)* measure please refer to the *minimum Q* parameter.

9 DEFINING TESTING PARAMETERS IN AQ21

9.1 Overview

AQ21, in addition to its learning capabilities, also provides a wide range of methods for testing learned hypotheses and classifying new examples according to the learned class descriptions. This section describes implemented methods of testing, and how the t*ests* component is organized in the AQ21 input file.

AQ21 provides a variety of ways to test learned hypotheses. The following sub-sections describe the testing algorithms that are implemented (ATEST, EPIC, EPIC-RB, and EPIC-P).

9.2 Structure of *Tests* component in the AQ21 input file

This component specifies the tests to be performed on the learned rulesets and the testing parameters to be used. The following paragraphs will describe in detail the parameters that are used for testing.

```
Example:

Tests
{
  examples
    {
      Method = ATEST
      Full_report = true
    }

  episodes
    {
      Method = EPIC
      Tolerance = 0.3
      Full_report = true
  }

}
```

The definition of the ***Tests*** component should follow the mandatory ***Runs*** component and the optional input hypotheses components (both for standard and prediction-based testing). Its definition starts with the keyword ***Tests***, followed by the tests description. The general form of the component is presented below.

```
Tests
{
  test1
    {
      test 1 parameters
    }
  …
  testn
    {
      test n parameters
    }
}
```

The number of tests defined in AQ21 is not limited, and depends only on the goals that the user wants to achieve. Definitions of all tests start with a unique test name followed by the test parameters.

9.3 Testing Parameters

This section describes in detail all parameters that can be used to define tests.

Evaluation of selector

Syntax:

Evaluation_of_selector = value

Possible values:

strict, flexible

Default value:

flexible

Example:

```
Evaluation_of_selector = strict
```

Comment:

The ATEST module in the AQ21 system has two possible ways to compute the degree of match between an event and a selector (condition): strictly of flexibly. The *strict* match returns 1 whenever an event is covered by a selector and 0 otherwise. The *flexible* match returns 1 whenever the event is covered by the selector and a value *d* smaller than 1 and greater or equal to 0 whenever the selector does not cover the event. For *continuous* and *discretized continuous* attributes, the value *d* decreases linearly with the distance between the interval specified in the selector and value from the event. For other types of

attributes the *flexible* selector match is equivalent to the *strict* selector match. Details if the matchind methods are described in (Michalski, 2004)

Evaluation of conjunction

Syntax:

Evaluation_of_conjunction = value

Possible values:

strict, coverage_ratio, selectors_ratio, flexible, min, min_w, prod, avg, avg_w

Default value:

strict

Example:

```
Evaluation_of_conjunction = selectors_ratio
```

Comment:

This parameter is used to select the method of evaluating degree of match of an event to conjunction of selectors (single rule).

Depending on the method used to evaluate selectors, AQ21 provides a number of different methods of evaluating conjunctions of selectors (rules). The strict selector match can be used with *strict*, *coverage_ratio*, *selectors_ratio*, and *flexible* methods of evaluating conjunctions, while the flexible selector match can be used with *min*, *min_w*, *prod*, *avg*, and *avg_w* methods of evaluating conjunctions (Michalski, 2004). The following paragraphs describe these methods.

Evaluation of conjunction of strict selector evaluations:

The *strict* match checks if the event is fully included in the rule (strictly matches all selectors in the rule). If yes, the value of degree of match is one, otherwise it is zero.

If the testing event fully matches the rule, the *coverage ratio* returns the ratio between the number of positive examples covered by the rule and the total number of positive training examples in the class. It returns zero otherwise. This method of evaluation of conjunction makes sense when a probabilistic sum is used for evaluation of disjunction (see below).

The *selectors ratio* measure returns the ratio between number of matched selectors to the total number of selectors in the rule.

The *flexible* match returns the ratio between the sum of matched selectors and the number of output attributes to the number of attributes in the event, separately for the continuous and discrete attributes. The program takes average of degrees of flexible match for the continuous and discrete attributes.

Evaluation of conjunction of flexible selector evaluations:

The *min* match returns the minimum of the degrees of match of the individual selectors in the rule.

The *min_w* match returns the degree of match of the selector that has the minimum degree of match weighted by its coverage. The weight of each selector is defined by the formula $w = p/(p+n)$, where w is weight of the selector, p is number of positive examples covered by the selector and n is the number of negative examples covered by the selector.

The *product* match returns the product of the degrees of match of all the selectors in the rule.

The *avg* match returns the average of the degrees of match of all the selectors in the rule.

The *avg_w* match returns the average of degrees of match of all selectors, weighted by the coverage of the selectors. Analogously to min_w method, the weight of each selector is defined by the formula $w = p/(p+n)$, where w is weight of the selector, p is number of positive examples covered by the selector and n is the number of negative examples covered by the selector.

Evaluation of disjunction

Syntax:

Evaluation_of_disjunction = value

Possible values:

average, prob_sum, max, best_only

Default value:

average

Example:

```
Evaluation_of_disjunction = prob_sum
```

Comment:

The evaluation of disjunction parameter is used to control how degrees of match for individual rules are combined into a degree of match for a ruleset. Since degree of match is a number in the range 0 to 1, each degree of match can be viewed as probability; the combination of such numbers must still be a degree of match (a number in range 0 to 1).

Average computes the average of degrees of match of rules. Similarly *prob_sum* uses probabilistic sum as the aggregation function. *Max* selects the highest degree of match. *Best_only* returns the degree of match from the best rule in the ruleset, according to the LEF sort criteria (not necessarily the highest degree of match).

.

Default class

Syntax:

Default class = class

Possible values:

Name of the run that represents the selected class.

Default value:

other

Example:

```
Default_class = good_robots
```

Comment:

If an event or episode could not be classified, because its degrees of match to all classes are below the acceptance threshold, it is assigned to the default class. The default class can be either one of classes or "other," which means that no classification is made. Please note that the program does not check if the class name is the actual name of one of the classes, but if not, AQ21 sets the classification as other.

Prior probability for "other" class

Syntax:

Prior_probability_other = value

Possible values:

A real number in range from 0 to 1.

Default value:

1 / (2 * number of classes + 1)

Example:

```
Prior_probability_other = 0.01
```

Comment:

EPIC-RB, the rule-Bayesian hybrid testing method assumes the prior probability of the "other" class to be the value of this parameter. The default, relatively small, value is defined as

*1 / (2 * number of classes + 1).*

Full report

Syntax:

Full_report = value

Possible values:

true or false

Default value:

false

```
Example:

    Full_report = true
```

Comment:

This parameter is used to specify how detailed the output from the testing module should be. The full report additionally contains information about the classification of each testing event. In cases when the testing dataset is large, the output is also large. Format of the output is described in Section 10.3.

Ignore events

Syntax:

Ignore_events = rules

Possible values:

a set of rules

```
Example:

    Ignore_events <-- [color = red, yellow][shape = square]
                  <-- [color = blue]
```

Comment:

This parameter is similar to one defined in the ***Runs*** *component*. It allows AQ21 to ignore events that match any of the specified rules. For more details, please refer to the ***Runs*** *component* description in Section 8.2 and definition on page 42.

Method

Syntax:

Method = value

Possible values:

atest, epic, epic_p, epic_rb

Default value:

atest

Example:

```
Method = epic
```

Comment:

This parameter is used to apply a basic method of testing. Testing can be performed for classification of single events (ATEST) or for classification of whole episodes (EPIC, EPIC-P, EPIC-BR). The following paragraphs will describe the testing methods in more detail.

ATEST

Atest is an event classification algorithm implemented in AQ21. The program matches a single event against rulesets for each class to generate a degree of match for the classes. Matching procedure is controlled by *evaluation of selector*, *evaluation of conjunction*, and *evaluation of disjunction* parameters described previously. Depending on parameters settings, it may provide a single or multiple classification (if the event matches more than one class with approximately the same degree or match – see the *tolerance* parameter description).

EPIC

Epic is an extension of the ATEST algorithm. Instead of classifying single events, EPIC classifies entire episodes (sequences of events that share the same value of the Episode meta-attribute). EPIC works by applying ATEST to all of the episode's events, and then aggregating their degrees of match. As can event classification in ATEST, EPIC can give a single or multiple classifications for the episode.

EPIC-P

Epic-P is an episode classification algorithm that classifies episodes according to the prediction-based classification algorithm.

EPIC-BR

This version of the Epic algorithm combines a standard episode classification algorithm that computes degrees of match between rulesets and events with a Bayesian

classification model. A degree of match of en event to class is computed as a combination of the degree generated by the *ATEST* module and a probability value from the Bayesian model.

Runs

Syntax:

Runs = list of runs

Possible values:

comma separated list of runs

Default value:

all runs

```
Example:

    Runs = good_robots, bad_robots
```

Comment:

This option is used to specify runs that are used for testing. The names of the runs used here need to be defined before in the *runs component*. Each run identifies a learned class or group of classes (in cases where * was used – see the *consequent* description in the *runs component* (page 36).

Stop when Decisive Advantage Probe

Syntax:

SDA_probe = value

Possible values:

integer greater than 1

Default value:

100

```
Example:

    SDA_probe = 30
```

Comment:

Stop when Decisive Advantage is an EPIC algorithm modification that allows classifications to be made on parts of episodes. When the degree of match of an episode

to one of the classes is clearly better than its degree of match to the second best class, EPIC-SDA ignores the rest of the examples from this episode (the classification is already indicative). The *sda_probe* parameter controls the minimum number of events from the testing episode that must be evaluated before the SDA condition will be applied.

Stop when Decisive Advantage Threshold

Syntax:

SDA_threshold = value

Possible values:

integer greater or equal 0

Default value:

0

```
Example:
    SDA_threshold = 2.5
```

Comment:

The *SDA_threshold* controls the minimum ratio between the best degree of match and the second best degree of match needed to stop evaluating new examples in the EPIS-SDA testing algorithm. If this value is below 1, the SDA method is not used.

Threshold

Syntax:

Threshold = value

Possible values:

real value in range from 0 to 1

Default value:

0

```
Example:
    Threshold = 0.3
```

Comment:

Threshold defines the minimum value of the degree of match for a classification to be accepted. If the degree of match of an event or an episode to a class is below the threshold, it will not be classified to this class. If the degrees of match are below the threshold for all classes, the event (or episode) is classified as *default class* (usually class "other").

Tolerance

Syntax:

Tolerance = value

Possible values:

real value in range from 0 to 1

Default value:

0

```
Example:
    Tolerance = 0.1
```

Comment:

Tolerance defines range of the degree of match that is used for multiple classifications. The testing event or episode is classified to a class if its degree of match to the class is both above the Threshold, and above the product of the maximum degree of match for the event/episode and the tolerance.

9.4 Input hypotheses

9.4.1 Overview

The AQ21 testing can be applied not only to the hypotheses learned during the same execution of the program, but also to those loaded from the input file. It is especially important feature when there is huge amount of data to learn from, which would be very time-consuming. AQ21 allows testing hypotheses previously learned and/or entered manually, by an expert in the domain.

9.4.2 Defining Input hypotheses

The ***Input_hypotheses*** component can be used to load hypotheses that will be examined by the ATEST, EPIC, and EPIC-RB testing methods. It contains definitions of classes, rules, and optionally additional information about the hypotheses.

For each class for which hypotheses are loaded from the input file, there must be a run component that corresponds to the class (has the same name). The learning mode in such run definition should be set to "test," meaning that no learning takes place, only testing.

The structure and description of the input hypotheses component is presented below.

```
Input_hypotheses class
{

positive_events = pos
negative_events = neg

consequent

rules
}
```

Class is a unique name of a class that is identical with the name of a run in the runs component.

Pos is the number of positive examples covered by the hypothesis.

Neg is the number of negative examples covered by the hypothesis.

Consequent is a complex that defines the class. For details please refer to description of consequent in the *runs component* (Section 8.3).

Rules is a list of rules, where each rule is denoted by a "<--" sign followed by a complex.

The *positive_events* and *negative_events* parameters are optional.

Example:

```
Input_hypotheses user1
{
  positive_events = 72
  negative_events = 12

[user = 1]
    <-- [host = 11, 13][app = outlook] : p = 60, n = 9
    <-- [hour = 7..10] : p = 10, n = 2
    <-- [app = acc, word][del=3.5] : p = 7, n = 3
}
```

For each class there must be exactly one ***Input_hypotheses*** *component*.

ATEST, EPIC, and EPIC-RB do not allow ambiguities in testing data. It means that each testing event has to match exactly one decision among the consequents of tested classes.

9.4.3 Defining input for Prediction-Based Model testing

The Prediction Based Model learns rules within a model without considering data for other models. During the testing phase, rules from all models are tested together and need to be loaded at the same time. Hypotheses for the Prediction-Based Model can be tested using only the EPIC-P method.

The structure of the prediction-based model input is presented below.

```
PBM_input
{
  model1
  {
    consequent = consequent1.1
        rules
    consequent = consequent1.2
        rules
    ...
  }

  model2
  {
    consequent = consequent2.1
        rules
    ...
  }
…
}
```

Model1, *model2*, … are complexes that define models. *Consequent1*.1, … are complexes that define classes within models. *Rules* is list of rules where each rule is "<--" sign followed by complex.

```
Example:

PBM_input
{
  [user = 1]
  {
     class = [shape = t]
         <-- [shape-3 = r]
         <-- [color-1 = r]
     class = [shape = s]
       <-- [shape-3 = t]
           [color-1 = g]
  }

  [user = 2]
  {
     class = [shape = r]
         <-- [color-2<>r]
  }

}
```

For details of the EPIC-P algorithm please refer to the description of the *Method* parameter described earlier in this document.

10 AQ21 OUTPUT

10.1 Overview

The AQ21 program generates output in a format that is identical to the one used the input files described in the previous sections. This is a very important feature of the program, since the same parameters can be reused and applied with new data, learned hypotheses can be manually modified and applied to testing data, or they can be updated when new data became available. The following sections describe the output from the learning and testing modules. An example of full AQ21 input and output is presented in Appendix A.

10.2 Output from learning

Similar to the input files, the AQ21 output starts with description, ***Common_domains***, and ***attributes*** components. The components are generated from the actual program settings, meaning that any default parameter values are also displayed. ***Common_domains*** and ***Attributes*** components may be different from the input file if the program has constructed new attributes or dropped existing ones. A description of all parameters in the ***Common_domains*** and ***Attributes*** components are presented in Sections 5 and 6.

Learning parameters are displayed in the same format as in the ***Runs*** component in the input file. This component contains all parameters used for learning - not only those set up by the user, but also the parameters for which default values were used. Such an output provides the user with complete information about the learning process, and can be used to reproduce experiments.

Example:

```
Common_domains
{
      color nominal  { r, y, b, g, w }
      shape linear  { r, s, t }
      user nominal  { 1, 2, 3 }
      # Derived Domains
      Count_shape linear  4
      Count_color linear  3
}


Attributes
{
      x1 shape epsilon = 0.5 cost = 1
      x2 shape epsilon = 0.5 cost = 1
      x3 shape epsilon = 0.5 cost = 1
      x4 color epsilon = 0.5 cost = 1
      x5 color epsilon = 0.5 cost = 1
      user user epsilon = 0.5 cost = 1
}

Runs
{
```

```
 Output_attributes = user
 Random_seed = 975313573

   user1
   {
     Consequent = [user=1]
     Learn_rules_mode = standard

     Maxstar = 2     Maxrule = 5     Ambiguity = positive
     Trim = Optimal
     Exceptions = false

     Mode = tf
     Minimum_u = 1

     Optimize_ruleset = true
     Continuous_optimization_probe = 5

     Truncate = true
     Display_selectors_coverage = true
     Display_values_coverage = false
     Display_events_covered = false
     Dompute_alternative_covers = false

     LEF_star
     {
         MaxNewPositives, 0.3
         MinNumSelectors, 0.3
         MinComplexity, 0.3
         MinCost, 0.3
     }

     LEF_partial_star
     {
         MaxNewPositives, 0
         MinNumSelectors, 0
         MinComplexity, 0
         MinCost, 0
     }

     LEF_sort
     {
         MaxPositives, 0
     }

   }
}
```

The most important part of the output is the ***Output_hypothesis*** component, which shows the results of the learning process. The structure of the ***Output_hypothesis*** component is identical to structure of the ***Input_hypothesis*** component presented in Section 9.4. It contains learned rules and additional information: learning time, number of rules in the cover, number of selectors (conditions) in the cover, complexity of the cover, average number of rules kept from each star, and number of uncovered positives.

Positive events and negative events are parameters that denote numbers of positive and negative events used for learning the target class respectively.

Each rule consists of the sign "<--" followed by a list of selectors and additional parameters. AQ21 uses a simplified form of the selectors defined in (Michalski, 2004), specifically:

[att rel val]

where *att* is the attribute name, *rel* is a relation, and *val* is a value, list of values, or a range. In addition, each selector may contain two numbers that represent the numbers of positive and negative events covered by the selector. The presence of the two numbers is controlled by the *Display_selectors_coverage* parameter described in Section 8.3. The list of selectors is followed by the symbol ":" followed by a comma-separated list of additional parameters. The list of additional parameters is presented in the table below. Parameters are shown only if their values are computed (for example when appropriate LEF criteria are selected). For example the value of *u* is not present in the output if the unique coverage is not computed for a rule.

p	number of positive examples covered by the rule
ep	estimated number of positive examples covered by the rule
np	number of positive examples covered by the rule and not covered by the previously learned rules
enp	number of positive examples covered by the rule and not covered by the previously learned rules
n	number of negative examples covered by the rule
en	estimated number of negative examples covered by the rule
q	value of q(*w*)
eq	estimated value of q(*w*)
npq	value of q(*w*) for new positives
enpq	estimated value of q(*w*) for new positives
u	unique coverage of the rule, i.e, the number of positive events covered by this rule and not covered by any other rules
cx	complexity of the rule
c	cost of the rule
s	significance of the rule
#	rule unique numeric identifier

Table 3: Parameters displayed with rules

```
Example:

Output_Hypotheses user1
{
  # -- This learning took =
  # -- System (CPU) time  = 2
  # -- User (Total) time  = 2
  # -- Number of rules in the cover = 1
  # -- Number of conditions         = 2
  # -- Complexity for this cover    = 14
  # -- Average number of rules kept from each stars = 3
  # -- Uncovered Positives = 0

  positive_events           = 4
  negative_events           = 2
[user=1]
       # Rule 1
   <-- [Count_shape_Eq_s=2 : 4,1]
       [Count_color_Eq_g=0 : 4,1]
        : p=4,np=4,u=4,cx=14,c=1,s=4 # 36


}
```

In the above example the hypothesis for "user=1" contains one rule with two selectors (conditions). It covers four positives and zero negatives, the unique coverage of the rule is four, its complexity is fourteen, its cost is one, and its significance is four. The rule's unique identifier is 36. The learning time for this hypothesis was two seconds, and the cover has two conditions (selectors).

10.3 Output from testing

Output from the testing module consists of two parts: a ***Tests*** component that contains parameters used in testing, and the actual result of testing that includes classifications that were made and some summaries. For a description of testing parameters please refer to Section 9. The example below presents the output of testing for a simple problem from the robots domain from the iAQ program (Michalski and Pietrzykowski, 2004). The ATEST program was used for testing (classification of events).

Example:

```
(#
# -- Testing: Atest

Class0 = robots_friendly
Class1 = robots_unfriendly

True   Event   Event   Event    Degrees of match
Class  Episode Number  Freq.  Class0         Class1   Assigned
Class(es)
0      0      14      1      1.0000       0.0000       { 0 }
0      0      15      1      1.0000       0.0000       { 0 }
0      0      16      1      1.0000       0.0000       { 0 }

1      0      17      1      0.0000       0.5000       { 1 }
1      0      18      1      0.0000       0.5000       { 1 }
1      0      19      1      0.0000       0.5000       { 1 }


       Class0  Class1  Other # of Examples
Class0      3      0      0      3
Class1      0      3      0      3
Other       0      0      0      0

Event Classification bars
# of decisions    Correct     Incorrect
1                 6           0
2                 0           0
Other             0           0


Total Events                              = 6
Definite Decisions                        = 6

Multiple Event Classifications            = 6
Correct Multiple Event Classifications    = 6
Predictive Accuracy Multiple              = 100.00%
Predictive Accuracy Averaged Multiple     = 100.00%
Precision Multiple                        = 100.00%

Single Event Classifications              = 6
Correct Single Event Classifications      = 6
Predictive Accuracy Single                = 100.00%
Predictive Accuracy Averaged Single       = 100.00%
Precision Single                          = 100.00%

Decision rate                             = 1.0000
Prediction gain                           = 2.0000

Test Method Type          = ATest
Evaluation of Selector    = strict
Evaluation of Conjunction = strict
Evaluation of Disjunction = average

tolerance = 0.0000
threshold = 0.0000
```

Test Method name

Mapping of class names

Description of Events (only in full_report mode)

Confusion Matrix

Event Class. Bars

Summary information about testing

Measures of the quality of classifications

Method's Parameters

```
This testing took:
0.0000 of System (CPU) time
0.0000 of User (Total) time

#-- end testing
#)
```

As shown in the above example all testing results are commented out, so the AQ21 output can be directly used also as input, and testing results will be ignored by the parser.

The testing output starts with the definition of classes and their correspondence to runs defined in the AQ21 ***Runs*** component. In the example above, class0 and class1 denote respectively *robots_friendly* and *robots_unfriendly* runs. The order of classes may or may not be the same as the order of definitions of runs; they are ordered lexicographically.

In the *full output* mode, the program displays classification information for all testing events (one row per testing event). The meaning of the columns describing those events is following: (1) true class (from testing data), (2) episode number (from testing data, used only by EPIC classifiers), (3) event number (unique event id also referred to as a key), (4) event frequency, (5) degrees of match to all classes, and (6) assigned classes. Because classification is based on degrees of match, the program may classify events to more than one class depending on the tolerance parameter (multiple classifications).

Confusion matrix defines how many events for true classes (rows) are classified to different classes (columns). In a perfect classification, positive values will be present only on the diagonal of the matrix, and zeros will be prevalent outside of the diagonal. The last column in the matrix shows the total number of events from each class. In case of multiple classifications, it will not simply be the sum of the other columns.

Event classification bars show how many times the program made classifications to one, two, three, etc. classes, and how many times the decision was correct or incorrect. Such information is important to evaluate the multiple classifications, and check how often AQ21 could not find a correct class within a given tolerance. In the example above, all events were correctly and precisely classified, as indicated by number 6 in the first row of classification bars.

Total events is the total number of events (examples) used in testing.

Definite decisions refers to how many times AQ21 assigns one class to an event. Note that it is not necessarily the number of events (for example when program makes multiple classifications).

Another section of the output from the testing module refers to multiple classifications. *Multiple Event Classification* and *Correct Multiple Event Classification* refer respectively to the total number and the number of correct multiple classifications. *Predictive Accuracy Multiple* is defined as the ratio of the number of the correctly classified examples to the total number of testing examples, even if the classification was not precise and the program has chosen more than one possible class. *Predictive Accuracy*

Averaged Multiple is defined as the average of predictive accuracies for all classes computed separately. It is computed using the formula:

$$avg(\frac{correct(i)}{total(i)})$$

where *correct(i)* is number of correctly classified examples for i-th class and *total(i)* is total number of testing examples from the i-th class. *Precision Multiple* is defined as:

$$\frac{\#events*\#classes-\#mclassifications}{\#mclassifications*(\#classes-1)}$$

if number of classes is greater than one, and

$$\frac{\#events}{\#mclassifications}$$

if number of classes is equal to one. In the formulas above, #events is the total number of test events, #classes is the total number of classes, and #mclassifications is the total number of multiple classifications.

The *Single Event Classification* section refers to the single classification of events (a.k.a. first choice classification). A single classification is made when the degree of match to a given class is the highest among degrees of match of all classes. In fact, even if an example was classified to more that one class it may contribute to single classification if its degree of match to the correct class is the strongest. *Single Event Classification*, *Correct Single Event Classification*, *Predictive Accuracy Single*, *Predictive Accuracy Averaged Single*, and *Precision Single* are defined similarly to the parameters for multiple classifications.

Decision rate is the ratio between the number of decisions (i.e., classifications to at least one class) to the total number of examples. *Prediction Gain* is defined as Prediction Accuracy Multiple multiplied by the number of classes, which corresponds to "how often the result is better than a random guess."

At the end of the testing output, AQ21 displays the parameters used for testing and information about system and user times that were required for the testing.

Additional output form EPIC classifiers

The EPIC classification program similarly displays information about the classification of episodes. An example of specific output from the EPIC program is presented below, where there are only sections corresponding to the episode classification - all sections common with the ATEST testing module are omitted (in the real output they are also present since EPIC applies ATEST for event classification).

Example: ***Classifications for the Episodes***	

.

	Class0	**Class1**	**Other**		*Confidence Matrix* (degrees of match of episodes to all classes)
Class0					
Episode1	*0.7500*	*0.0000*	*0.1250*		
Episode2	*0.7500*	*0.0000*	*0.1250*		
Class1					
Episode3	*0.2500*	*0.7500*	*0.0000*		
Episode4	*0.0000*	*0.2500*	*0.3750*		
Other					
	Class0	**Class1**	**Other**	**# of Episodes**	*Confusion Matrix* (numbers of episodes classified to particular class)
Class0	*2*			*2*	
Class1		*1*	*1*	*2*	
Other				*0*	
Epic Episode Classification bars					*Episode Classification bars*
# of decisions	**Correct**	**Incorrect**			
1	*3*	*1*			
2	*0*	*0*			
Other	*0*	*0*			
Epic Total Episodes				*= 4*	Summary information about testing
Epic Definite Decisions				*= 4*	
Epic Multiple Episode Classifications				*= 4*	Measures of the quality of classifications
Epic Correct Multiple Episode Classifications				*= 3*	
Epic Predictive Accuracy Multiple				*= 75.00%*	
Epic Predictive Accuracy Averaged Multiple				*= 75.00%*	
Epic Precision Multiple				*= 100.00%*	
Epic Single Episode Classifications				*= 4*	
Epic Correct Single Episode Classifications				*= 3*	
Epic Predictive Accuracy Single				*= 75.00%*	
Epic Predictive Accuracy Averaged Single				*= 75.00%*	
Epic Precision Single				*= 100.00%*	
Epic Decision rate	*= 1.0000*				
Epic Prediction gain	*= 1.2500*				
Epic Classification gain	*= 0.0000*				

In the presented example, all sections have analogous meanings to the corresponding sections for individual event classification. The degree of match of an episode to a given class is defined as the average of degrees of match of all events from the episode to the class.

In the EPIC section of the testing output, AQ21 first shows a *Confidence Matrix* that contains the degrees of match between episodes and classes, and then a *Confusion Matrix* that contains the numbers of episodes classified to particular classes. The next sections of the EPIC output contain summaries (*Accuracy*, *Precision* etc.) that are analogous to the ATEST summaries. The only difference between the ATEST and EPIC outputs is that all summaries in EPIC are based on the classification of episodes rather than individual events.

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.

Bloedorn, E. and Michalski R.S., "Constructive Induction from Data in AQ17-DCI: Further Experiments," *Reports of the Machine Learning and Inference Laboratory*, MLI 91-12, School of Information Technology and Engineering, George Mason University, Fairfax, VA, December, 1991.

Bloedorn, E., Wnek, J. and Michalski R.S., "Multistrategy Constructive Induction: AQ17-MCI," *Reports of the Machine Learning and Inference Laboratory*, MLI 93-4, School of Information Technology and Engineering, George Mason University, May, 1993.

Clark, P. and Niblett T., "The CN2 Induction Algorithm," *Machine Learning*, 3(4):261-283, 1989.

Glowinski, C. and Michalski R.S., "Discovering Multi-head Attributional Rules in Large Databases," *Tenth International Symposium on Intelligent Information Systems*, Zakopane, Poland, June, 2001.

Kaufman K., "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, November, 1997.

Kaufman K. and Michalski R.S., "The Development of the Inductive Database System VINLEN: A Review of Current Research," *International Intelligent Information Processing and Web Mining Conference*, Zakopane, Poland, 2003.

Kerber, R. "Chimerge: Discretization for Numeric Attributes." *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI '92)*, AAAI Press, pp. 123-128, 1992.

Michalewicz, Z., "Genetic algorithms + data structures = evolution programs," Springer-Verlag New York, 1994

Michalski R.S., "On the Quasi-Minimal Solution of the General Covering Problem," *Proceedings of the V International Symposium on Information Processing (FCIP 69)(Switching Circuits), Vol. A3*, Bled, Yugoslavia, pp. 125-128, October 8-11, 1969.

Michalski R.S., "ATTRIBUTIONAL CALCULUS: A Logic and Representation Language for Natural Induction," *Reports of the Machine Learning and Inference Laboratory*, MLI 04-2, George Mason University, Fairfax, VA, April, 2004.

Michalski R.S. and Kaufman, K., "The AQ19 System for Machine Learning and Pattern Discovery: A General Description and User's Guide," *Reports of the Machine Learning and Inference Laboratory*, MLI 01-2, George Mason University, Fairfax, VA, 2001.

Michalski R.S. and Kaufman K., "Learning Patterns in Noisy Data: The AQ Approach," *Machine Learning and its Applications*, Paliouras, G., Karkaletsis, V. and Spyropoulos, C. (Eds.), pp. 22-38, Springer-Verlag, 2001b.

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., "Incremental Generation of VL1 Hypotheses: The Underlying Methodology and the Description of Program AQ11," *Reports of the Intelligent Systems Group*, ISG 83-5, UIUCDCS-F-83-905, Department of Computer Science, University of Illinois, Urbana, January 1983.

Michalski, R.S and Pietrzykowski, J., "iAQ: A Natural Induction System for Education and Research in Machine Learning and Knowledge Mining," *Reports of the Machine Learning and Inference Laboratory*, George Mason University, Fairfax, VA, 2004 (to appear).

Michalski, R.S and Wojtusiak, J., "Reasoning with Meta-values in AQ Learning," *Reports of the Machine Learning and Inference Laboratory*, MLI 05-1, George Mason University, Fairfax, VA, 2005.

Quinlan, J. R., *C4.5 Systems for Machine Learning*, Morgan Kaufmann Publishers Inc., 1993.

Witten, H. and Frank, E., *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann, October, 1999.

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, March ,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.

The AQ21 and iAQ programs can be downloaded from The Machine Learning and Inference Laboratory website: *http://www.mli.gmu.edu/msoftware.html*

APPENDIX A APPLICATION OF AQ21 TO EXAMPLE PROBLEMS

This appendix shows an example of an AQ21 input file, AQ21 execution and program output. For ease of reading, in the presented example all comments are *italics* and all names of components are ***bold italics***. In the real AQ21 input file no formatting is allowed (a text file).

This example is taken from the iAQ program and uses the toy problem domain based on the classification of robots into friendly and unfriendly groups.

```
Description
{
This is an example of robots from the iAQ program. It shows basic parameter
settings of AQ21. In this example AQ21 learns rules discriminating between
friendly and unfriendly robots. It is done by learning the two classes
separately. After learning AQ21 applies the ATEST program to test learned
hypotheses.

All lines starting with # sign are comments, and are not interpreted by AQ21.
}

# This component defines common domains that can be used to define attributes
# that appear in data. Please note that this is optional component of the
# input file and attributes' domains can be defined explicitly in attributes
# component. Section 4 of this manual describes definition of domains in
# AQ21.
# Please also note that the presented example uses only nominal attributes,
# while AQ21 supports wide range of different types.
Common_domains
{
        color nominal { red, yellow, blue, green, black, white }
        shape nominal { round, square, triangle }
        height linear { short, medium, tall }
        inhand nominal { sword, balloon, flag, us_flag, polish_flag }
        bool nominal { no, yes }
}

# This component defines attributes that appear in data and are used for
# learning and testing. Each attribute corresponds to exactly one column
# in the data (rows in data are events/examples). To ignore
# certain columns use key word "ignore" instead of name and definition of an
# attribute. Definition of attributes in AQ21 is described in section 5 of
# this manual.
# Please note that this component of input file is mandatory.
Attributes
{
    head     shape            cost = 1
    body     shape            cost = 1
    smile    bool             cost = 1
    holding  inhand           cost = 2
    height   height           cost = 1
    antenna  color            cost = 3
    jacket   color            cost = 1
    tie      bool             cost = 2
```

```
    group    nominal { friendly, unfriendly }

}

# This component of the input file defines classes and learning parameters.
# The runs component is split into two parts, one defining global parameters
# for all runs and run-specific components that define specific parameters
# for each class to be learned. AQ21 requires definition of at least one
# class. Please note the runs component is mandatory. For details please
# refer to section 8 of this manual.
Runs
{
    # definition of global parameters
    Attribute_selection_method = promise

    # Definition of class robots-friendly
    robots-friendly
    {
       # Learn rules in Theory Formation (TF) mode in which learned cover
       # is complete and consistent.
       Mode = TF

       # Consequent of the class is defined as "group is friendly" what
       # means that all events with value of attribute group equal friendly
       # will be used as positive examples for learning. All other events
       # will be used as negative examples.
       Consequent = [ group = friendly ]

       # LEF Partial Star (LEF_PS) defines criteria of selection partially
       # learned rules during the star generation process. In this case three
       # criteria are used to select the best rules: maximum number of
       # positive examples with tolerance 0.1, minimum number of selectors
       # with tolerance 0, and minimum cost with tolerance 0. LEF_PS need not
       # be defined and its default value is presented in section 8.3.
       lef_ps
         {
           MaxNewPositives, 0.1
           MinNumSelectors, 0
           MinCost, 0
         }

       # AQ21 can learn and display alternative rulesets for the
       # learned class.
       Compute_alternative_covers = true
    }

    # Definition of class robots-unfriendly
    # Please note that some parameters like definition of LEF are not
    # specified and AQ21 uses their default values.
    robots-unfriendly
    {
       # Learn in Theory formation Mode
       Mode = tf

       # Define positive events as those that have value of group
       # equal unfriendly
       Consequent = [ group = unfriendly ]
```

```
       # Do not display alternative covers
       Compute_alternative_covers = false
    }
# end of runs component
}

# This component of the input file defines tests of learned hypotheses to be
# performed. Each subcomponent corresponds to one test (in this case the
# total number of tests is two). If no tests are defined, the entire tests
# component should be omitted in the input file. Within each test
# subcomponent at least one parameter needs to be specified. For details
# please refer to Section 8 of this manual.
Tests
{
   # Definition of the first test
   robots-test1
   {
      # Method of testing is ATEST (event classification program)
      Method = ATEST

      # Defines method of matching events against rules. In this case the
      # strict matching is applied to match events against rules.
      # If an testing event matches all selectors in a rule, degree of match
      # is 1, otherwise degree of match is 0.
      Evaluation_of_conjunction = strict

      # Defines method of evaluating degrees of match for rulesets. Here
      # degree of match of a testing event against a ruleset is computed
      # as maximum of degrees of match of all rules from this ruleset.
      Evaluation_of_disjunction = max

      # In full report mode program displays classification for every single
      # event. Please note that for larger datasets output from program may
      # also be very large.
      Full_report = true
   }

   # Definition of the second test
   robots-test2
   {
      # Selectors ratio method is used to compute degree of match of an
      # single testing event and a rule.
      Evaluation_of_conjunction = selectors_ratio

      # Maximum is used to aggregate degrees of match of single rules into
      # degree of match of entire ruleset.
      Evaluation_of_disjunction = max

      # Tolerance defines range of degree of match within which events
      # are classified. In this case program will classify to all classes
      # within 0.1 tolerance.
      Tolerance = 0.1

      # Display classification for every single event.
      Full_report = true
   }
# end of tests component
}
```

```
# The events component consists of a list of training and testing events. In
# this case it consists of only training examples because testing
# examples are in specified in a separate component.
# Each event consists of a list of comma-separated values of attributes. The
# order and number of attributes must be the same as defined in the
# attributes component. There is no comma after the last value in the event.
# Events are required by AQ21, however they may be specified in separate
# file, not necessarily in the events component of the input file.
Events
{
triangle, square,   yes, flag,        medium, green, blue,   yes, friendly
round,    square,   yes, flag,        tall,   green, blue,   yes, friendly
round,    triangle, yes, balloon,     medium, green, yellow, no,  friendly
square,   square,   yes, balloon,     short,  red,   yellow, no,  friendly
round,    triangle, yes, us_flag,     medium, green, yellow, no,  friendly
round,    triangle, yes, polish_flag, medium, green, yellow, no,  friendly

triangle, square,   no,  us_flag, medium,  yellow, blue,   yes, unfriendly
round,    square,   yes, sword,   medium,  green,  blue,   yes, unfriendly
round,    square,   no,  balloon, medium,  yellow, red,    yes, unfriendly
square,   square,   no,  balloon, medium,  red,    green,  yes, unfriendly
square,   triangle, yes, sword,   short,   green,  yellow, no,  unfriendly
round,    triangle, no,  flag,    short,   green,  black,  yes, unfriendly
square,   square,   yes, sword,   tall,    red,    red,    yes, unfriendly
}

# This component defines events used for testing. Format of events is the
# same as in the events component. Testing events can be specified either in
# this component or in a separate file. Testing events are optional.
Testing_events
{
triangle, square,   yes, balloon, medium, green,  blue, no,  friendly
round,    triangle, yes, flag,    short,  blue,   red,  yes, friendly
triangle, triangle, yes, us_flag, short,  yellow, red,  yes, friendly

square, square,   no,  us_flag, short,  blue,  yellow, no,  unfriendly
round,  triangle, no,  flag,    short,  green, black,  yes, unfriendly
square, round,    yes, sword,   medium, red,   red,    yes, unfriendly
}
```

AQ21 executed on input file presented above generates output presented below. Output from the program has the same format as the input file, but contains more information. The unshaded comments in *italics* were added manually to explain the output. Section 10 of this manual describes details of the AQ21 output.

```
# -- Opening file robots1.aq21
# -- Done Parsing
# -- Number of Events: 13
# -- Number of Testing Events: 6
# Creating file
# -- Output Generated by AQ21

The description component is copied directly from the input file.
Description
{
```

```
definition_filename = robots1.aq21


This is an example of robots from the iAQ program. It shows basic parameter
settings of AQ21. In this example AQ21 learns rules discriminating between
friendly and unfriendly robots. It is done by learning the two classes
separately. After learning AQ21 applies the ATEST program to test learned
hypotheses.

All lines starting with # sign are comments, and are not interpreted by AQ21.

}
```

The debug information is displayed for technical purposes only and should be ignored by users. It is used to identify the version of the program, and needs to be used when submitting any problems.

```
(#  -- Debugging information

The following information are required when submitting a bug.
They specify which version of the program was used.

$Id: aq21.cpp,v 1.8 2004/06/30 20:32:35 jwojt Exp $
$Id: AqLearner.cpp,v 1.14 2004/08/13 22:49:39 jwojt Exp $
$Id: BasicComplex.cpp,v 1.7 2004/08/12 22:53:11 jwojt Exp $
$Id: LEF.cpp,v 1.1.1.1 2003/06/18 21:07:23 jwojt Exp $
$Id: LEFCriterion.cpp,v 1.2 2004/08/12 22:53:11 jwojt Exp $
$Id: Domain.cpp,v 1.7 2004/06/30 20:32:12 jwojt Exp $
$Id: Attribute.cpp,v 1.8 2004/06/30 20:32:12 jwojt Exp $
$Id: Tests.cpp,v 1.10 2004/06/30 20:32:12 jwojt Exp $
$Id: CI.cpp,v 1.5 2004/05/10 23:11:14 jwojt Exp $

#)
```

In the domains component, AQ21 displays all domains used by the program. This includes domains defined by user and domains of derived attributes (please refer to Constructive Induction section). Please note that the domain for attribute "group" defined explicitly in the attributes component was displayed with the other domains.

```
Common_domains
{
      bool nominal  { no, yes }
      color nominal  { red, yellow, blue, green, black, white }
      group nominal  { friendly, unfriendly }
      height linear  { short, medium, tall }
      inhand nominal  { sword, balloon, flag, us_flag, polish_flag }
      shape nominal  { round, square, triangle }
}
```

The attributes component displays all attributes used in learning and testing. This includes derived attributes built by the Constructive Induction module.

```
Attributes
{
      head shape epsilon = 0.5 cost = 1
      body shape epsilon = 0.5 cost = 1
```

```
      smile bool epsilon = 0.5 cost = 1
      holding inhand epsilon = 0.5 cost = 2
      height height epsilon = 0.5 cost = 1
      antenna color epsilon = 0.5 cost = 3
      jacket color epsilon = 0.5 cost = 1
      tie bool epsilon = 0.5 cost = 2
      group group epsilon = 0.5 cost = 1
}
```

The runs component is copied from the input file, but all relevant AQ21 parameters are displayed. This guarantees that all results can be reproduced. For a description of displayed parameters please refer to section 8 of the manual.

```
Runs
{
 random_seed = 975313573

   robots-friendly
   {
     consequent = [group=friendly]
     learn_rules_mode = standard

     maxstar = 2     maxrule = 5     ambiguity = IncludeInPos
     trim = Optimal
     exceptions = false

     mode = tf
     minimum_u = 1

     optimize_ruleset = true
     continuous_optimization_probe = 5

     truncate = true
     display_selectors_coverage = true
     display_values_coverage = false
     display_events_covered = false
     compute_alternative_covers = true
     max_alternatives = 10

     LEF_star
     {
         MaxNewPositives, 0.3
         MinNumSelectors, 0.3
         MinComplexity, 0.3
         MinCost, 0.3
     }

     LEF_partial_star
     {
         MaxNewPositives, 0.1
         MinNumSelectors, 0
         MinCost, 0
     }

     LEF_sort
     {
         MaxPositives, 0
```

```
    }
  }

  robots-unfriendly
  {
    consequent = [group=unfriendly]
    learn_rules_mode = standard

    maxstar = 2     maxrule = 5     ambiguity = IncludeInPos
    trim = Optimal
    exceptions = false

    mode = tf
    minimum_u = 1

    optimize_ruleset = true
    continuous_optimization_probe = 5

    truncate = true
    display_selectors_coverage = true
    display_values_coverage = false
    display_events_covered = false
    compute_alternative_covers = false

    LEF_star
    {
        MaxNewPositives, 0.3
        MinNumSelectors, 0.3
        MinComplexity, 0.3
        MinCost, 0.3
    }

    LEF_partial_star
    {
        MaxNewPositives, 0
        MinNumSelectors, 0
        MinComplexity, 0
        MinCost, 0
    }

    LEF_sort
    {
        MaxPositives, 0
    }

  }

}
```

Similar to the runs component, the tests component is displayed to the output. All parameters, including those whose default values are used, are printed.

```
Tests
{
      robots-test1
      {
          evaluation_of_selector    = flexible
```

```
          evaluation_of_conjunction = strict
          evaluation_of_disjunction = max
          method      = ATest
          tolerance   = 0
          threshold   = 0
          full_report = true
          sda_probe       = 100
          sda_threshold   = 0
      }

      robots-test2
      {
          evaluation_of_selector    = flexible
          evaluation_of_conjunction = selectors_ratio
          evaluation_of_disjunction = max
          method      = ATest
          tolerance   = 0.1
          threshold   = 0
          full_report = true
          sda_probe       = 100
          sda_threshold   = 0
      }

}
```

The output hypotheses components are used to display learned rules and additional information generated during the learning process. For the class robots-friendly the alternative covers were generated (as requested in the input file) and one generated alternative is displayed. Additional information including number of positive and negative examples, complexity of cover etc. is also displayed. For a detailed description please refer to Section 10.2.

```
Output_Hypotheses robots-friendly
{
  # -- This learning took =
  # -- System (CPU) time  = 0
  # -- User (Total) time  = 0
  # -- Number of rules in the cover = 1
  # -- Number of conditions         = 2
  # -- Complexity for this cover    = 15
  # -- Average number of rules kept from each stars = 4
  # -- Uncovered Positives = 0

  positive_events          = 6
  negative_events          = 7
```

This part of output refers to alternative covers.

```
  # -- 1 Rule(s) Eliminated because not necessary
  # -- [smile=yes] [holding<>sword] [jacket=yellow,blue,white]
  : np=6,cx=26,c=1.33,s=5 # 51

(# Alternative ruleset 1
    Number of rules      = 1
    Number of conditions = 3
    Complexity           = 26
```

```
[group=friendly]
   <-- [holding<>sword] [antenna=red,green] [jacket=yellow,blue]
  : p=6,np=6,cx=26,c=2,s=6 # 61
#)
```

And finally the learned hypotheses. In this case there is one rule that covers all six positive examples and none negative examples. The rule has two selectors with attributes smile and holding respectively. For a description of the parameters displayed with the rules, please refer to Section 10.2.

```
[group=friendly]
       # Rule 1
   <-- [smile=yes : 6,3]
       [holding<>sword : 6,4]
        : p=6,np=6,u=6,cx=15,c=1.5,s=6 # 60


}
```

The class robots-unfriendly was learned without alternative covers (program default), so the only rules displayed are from the selected best cover. This cover has two rules, the first covering four, and the second covering three positive examples.

```
Output_Hypotheses robots-unfriendly
{
  # -- This learning took =
  # -- System (CPU) time  = 0
  # -- User (Total) time  = 0
  # -- Number of rules in the cover = 2
  # -- Number of conditions         = 2
  # -- Complexity for this cover    = 14
  # -- Average number of rules kept from each stars = 1
  # -- Uncovered Positives = 0

  positive_events           = 7
  negative_events           = 6
[group=unfriendly]
       # Rule 1
   <-- [smile=no : 4,0]
        : p=4,np=4,ep=4,n=0,en=0,u=4,cx=7,c=1,s=4 # 160

       # Rule 2
   <-- [holding=sword : 3,0]
        : p=3,np=3,u=3,cx=7,c=2,s=3 # 161


}
```

Output from testing is described in Section 10.3. In the presented case, two tests were applied: robots_test1 and robots_test2. In both tests, all testing events were classified correctly.

```
(#
# -- Testing: robots-test1

Class0 = robots-friendly
Class1 = robots-unfriendly
```

This part of the output is generated when the Full_report parameter is used. It shows event-by-event degrees of match to all classes and lists of classes to which each event was classified.

True Class	Event Episode	Event Number	Freq.	Degrees of match Class0	Class1	Assigned Class(es)
0	0	14	1	1.0000	0.0000	{ 0 }
0	0	15	1	1.0000	0.0000	{ 0 }
0	0	16	1	1.0000	0.0000	{ 0 }
1	0	17	1	0.0000	1.0000	{ 1 }
1	0	18	1	0.0000	1.0000	{ 1 }
1	0	19	1	0.0000	1.0000	{ 1 }

The matrix below shows the number of events classified to different classes. Rows represent real classes and columns represent classes to which events were classified. In a perfect classification (like in this example) all nonzero numbers are in diagonal.

	Class0	Class1	Other	# of Examples
Class0	3	0	0	3
Class1	0	3	0	3
Other	0	0	0	0

Event Classification bars

# of decisions	Correct	Incorrect
1	6	0
2	0	0
Other	0	0

```
Total Events                           = 6
Definite Decisions                     = 6

Multiple Event Classifications         = 6
Correct Multiple Event Classifications = 6
Predictive Accuracy Multiple           = 100.00%
Predictive Accuracy Averaged Multiple  = 100.00%
Precision Multiple                     = 100.00%

Single Event Classifications           = 6
Correct Single Event Classifications   = 6
Predictive Accuracy Single             = 100.00%
Predictive Accuracy Averaged Single    = 100.00%
Precision Single                       = 100.00%

Decision rate                          = 1.0000
Prediction gain                        = 2.0000

Test Method Type          = ATest
Evaluation of Selectors   = flexible
Evaluation of Conjunction = strict
Evaluation of Disjunction = max

SDA probe     = 100
```

```
SDA threshold = 0.0000

tolerance = 0.0000
threshold = 0.0000

This testing took:
0.0000 of System (CPU) time
0.0000 of User (Total) time

#-- end testing
# -- Testing: robots-test2

Class0 = robots-friendly
Class1 = robots-unfriendly

True  Event Event Event Degrees of match
Class Episode     Number      Freq. Class0      Class1      Assigned
Class(es)
0     0           14          1     1.0000      0.0000      { 0 }
0     0           15          1     1.0000      0.0000      { 0 }
0     0           16          1     1.0000      0.0000      { 0 }

1     0           17          1     0.5000      1.0000      { 1 }
1     0           18          1     0.5000      1.0000      { 1 }
1     0           19          1     0.5000      1.0000      { 1 }




      Class0      Class1      Other # of Examples
Class0      3     0     0     3
Class1      0     3     0     3
Other 0     0     0     0


Event Classification bars
# of decisions    Correct     Incorrect
1           6     0
2           0     0
Other       0     0

Total Events                          = 6
Definite Decisions                    = 6

Multiple Event Classifications        = 6
Correct Multiple Event Classifications = 6
Predictive Accuracy Multiple          = 100.00%
Predictive Accuracy Averaged Multiple  = 100.00%
Precision Multiple                    = 100.00%

Single Event Classifications          = 6
Correct Single Event Classifications   = 6
Predictive Accuracy Single            = 100.00%
Predictive Accuracy Averaged Single    = 100.00%
Precision Single                      = 100.00%

Decision rate                         = 1.0000
Prediction gain                       = 2.0000
```

```
Test Method Type          = ATest
Evaluation of Selectors   = flexible
Evaluation of Conjunction = selectors_ratio
Evaluation of Disjunction = max

SDA probe     = 100
SDA threshold = 0.0000

tolerance = 0.1000
threshold = 0.0000

This testing took:
0.0000 of System (CPU) time
0.0000 of User (Total) time

#-- end testing

#)
```

.

APPENDIX B COMPARISON OF FEATURES OF C4.5, AQ19, AND AQ21

Feature:	C4.5	AQ19	AQ21	Comment
Attribute types				
Nominal	Y	Y	Y	
Linear		Y	Y	
Cyclic		Y		
Discretized Continuous		Y	Y	
Integer			Y	
Continuous	Y	Y	Y	
Structured nominal		Y	Y	
Structured linear				
Set-valued				
Compound				
Output types				
Single attribute (selected or all values)- nominal	Y	Y	Y	
Structured attribute		Y		
Sequential covers		Y		
Multi-head			Y*	* Definition of class is a complex that may have more than one attribute. Program cannot compute automatically Cartesian product of domains of output attributes to learn all multi-head classes.
Order of selectors		Y		
Alternative covers		Y*	Y**	* Implemented in special version used for iAQ ** Based on maxrule
Negation of selectors		Y*	Y**	* Implemented in special version used for iAQ ** Negation ratio is 0.6
Special values				
Do not know "?"	Y		Y	
Not applicable "N/A"		Y	Y	
Irrelevant "*"		Y	Y	
Running Program				
Batch mode	Y	Y	Y	
C4.5 input format	Y		Y	
AQ19 input format		Y	Y*	* Can be converted to AQ21 format by an external program
AQ21 input format			Y	
Input data processing				
Random event selection			Y	
PROMISE based attribute quality			Y*	* Used for attribute selection
Gain ratio based attribute quality	Y*		Y**	* Used for building decision trees ** Used for attribute selection
ChiMerge for automatic discretization			Y	
Automatic domain discovery				
Ambiguities		Y*	Y**	* IncludeInPos, IncludeInNeg, IncludeInMajority, Ignore, Max ** IncludeInPos, IncludeInNeg, IncludeInMajority, Ignore
Ignoring events			Y*	* Rules that define events to be ignored. Can be ignored for all

				classes or a specific class.
Intelligent Target Dataset Generator (ITG)*				* Implemented in INLEN and VINLEN systems
Learning Modes				
Theory Formation (TF)		Y	Y	
Approximate Theory Formation (ATF)			Y*	* Learns rules like in TF mode, but optimizes them as in PD mode. Does not guarantee neither completeness nor consistency.
Pattern Discovery (PD)		Y	Y	
Uniclass		Y		
Incremental Learning				
Full Event Memory*				* Implemented in INLEN system
Partial Event Memory*				* Implemented in AQ11 system
No Event Memory		Y		
Other Learning Params				
Trim		Y	Y*	* MostSpec, Optimal, MostGen
Multi-seed star generation			Y	
Rule/ruleset optimization				
Truncate rules		Y	Y	
No-loss truncation		Y		
LEF Criteria*				* AQ19 has additional criteria not listed here
MaxPositives		Y	Y	
MaxEstimatedPositives			Y	
MinNegatives		Y	Y	
MinEstimatedNegatives			Y	
MaxNewPositives		Y	Y	
MaxEstimatedNewPositives			Y	
MaxQ		Y	Y	
MaxEstimatedQ			Y	
MaxNewPositivesQ			Y	
MaxEstimatedNewPositivesQ			Y	
MaxNumSelectors		Y	Y	
MinNumSelectors		Y	Y	
MaxSignificance			Y	
MinComplexity			Y	
MinCost		Y	Y	
GainRatio		Y	Y	
Constructive Induction				
A-rules			Y	
L-rules				
DCI			Y	
HCI				
KCI - Advise			Y	
Counting Attributes			Y	
Testing				
ATEST	Y	Y	Y	
EPIC		Y	Y	
EPIC-RB			Y	
EPIC-P			Y	
EPIC-SDA			Y	

APPENDIX C TEMPLATES

Template 1 Learning

Description

```
{
Template for learning problem. Please comment out not needed parts of the
file by using # and/or (#  #).
}
```

Domains

```
{
  color nominal {red, green, blue}
}
```

Attributes

```
{
background color
number linear { 0, 1, 2 }
length continuous 0, 200
class nominal {c1, c2}
}
```

Runs

```
{

  Run_c1
  {
  Mode = TF                   # Possible values: TF, ATF, and PD
  Consequent = [class=c1]
  Ambiguity = IncludeInPos    # Possible values: IncludeInPos, IncludeInNeg,
                              #   IgnoreForLearning, and DisplayAmbiguities
  Trim = Optimal              # Possible values: MostGen, Optimal, MostSpec
  Compute_alternative_covers = True
  Maxstar = 1
  Maxrule = 10
  }

  Run_c2
  {
  Mode = TF                   # Possible values: TF, ATF, and PD
  Consequent = [class=c2]
  Ambiguity = IncludeInPos    # Possible values: IncludeInPos, IncludeInNeg,
                              #   IgnoreForLearning, and DisplayAmbiguities
  Trim = Optimal              # Possible values: MostGen, Optimal, MostSpec
  Compute_alternative_covers = False
  Maxstar = 1
  Maxrule = 1
  }

  Run_All_in_PD
  {
  Mode = PD                   # Possible values: TF, ATF, and PD
  Consequent = [class=*]      # '*' indicates learning for all possible values
  Ambiguity = IncludeInPos    # Possible values: IncludeInPos, IncludeInNeg,
                              #   IgnoreForLearning, and DisplayAmbiguities
  Trim = Optimal              # Possible values: MostGen, Optimal, MostSpec
  Compute_alternative_covers = False
  Maxstar = 1
  Maxrule = 1
  }
```

```
  Run_Multi-head
  {
  Mode = PD                          # Possible values: TF, ATF, and PD
  Consequent = [class=c1][length<=40] # Definition of Multi-head rules
  Ambiguity = IncludeInPos      # Possible values: IncludeInPos, IncludeInNeg,
                                #   IgnoreForLearning, and DisplayAmbiguities
  Trim = Optimal      # Possible values: MostGen, Optimal, MostSpec
  Compute_alternative_covers = False
  Maxstar = 1
  Maxrule = 1
  }


}
```

Events

```
{
red, 1, 34.6, c1
green, 0, 2.45, c2
red, 1, 33.0, c1
blue, 0, 33.5, c2
}
```

Template 2 Learning and Testing

Description

```
{
Template for learning and testing problem. Please comment out not needed parts
of the
file by using # and/or (#  #).
}
```

Domains

```
{
  color nominal {red, green, blue}
}
```

Attributes

```
{
background color
number linear { 0, 1, 2 }
length continuous 0, 200
class nominal {c1, c2}
}
```

Runs

```
{

  Run_All_in_PD
  {
  Mode = PD                  # Possible values: TF, ATF, and PD
  Consequent = [class=*]     # '*' indicates learning for all possible values
  Ambiguity = IncludeInPos   # Possible values: IncludeInPos, IncludeInNeg,
                             #   IgnoreForLearning, and DisplayAmbiguities
  Trim = Optimal             # Possible values: MostGen, Optimal, MostSpec
  Compute_alternative_covers = False
  Maxstar = 1
  Maxrule = 1
  }
```

```
}

Tests
{
  Test1
    {
      Method = ATest         # Possible Values: ATest, EPIC, EPIC_P, EPIC_RB
      Evaluation_of_selector = Strict    # Possible values: Strict, Flexible
      Evaluation_of_conjunction = Strict # Possible values: Strict,
                                         #   Coverage_ratio,
                                         #   Selectors_ratio, Flexible, Min,
                                         #   Min_w, Prod, Avg, Avg_w
      Evaluation_of_disjunction = Max    # Possible values: Max, Prob_sum,
                                         #   Average, Best_only
      Default_class = Other              # Specity name of run or 'Other'
      Full_report = False
      Tolerance = 0.1
      Threshold = 0.1
    }

  Test2
    {
      Method = ATest         # Possible Values: ATest, EPIC, EPIC_P, EPIC_RB
      Evaluation_of_selector = Strict      # Possible values: Strict, Flexible
      Evaluation_of_conjunction = Selectors_ratio # Possible values: Strict,
                                                  #   Coverage_ratio,
                                                  #   Selectors_ratio,
                                                  #   Flexible, Min,
                                                  #   Min_w, Prod, Avg, Avg_w
      Evaluation_of_disjunction = Max    # Possible values: Max, Prob_sum,
                                         #   Average, Best_only
      Default_class = Other       # Specity name of run or 'Other'
      Full_report = False
      Tolerance = 0.1
      Threshold = 0.1
    }

}

Events
{
red, 1, 34.6, c1
green, 0, 2.45, c2
red, 1, 33.0, c1
blue, 0, 33.5, c2
}

Testing_events
{
blue, 1, 34.6, c1
blue, 0, 9.999, c2
}
```

Template 3 Testing

```
Description
{
Template for testing problem. Please comment out not needed parts of the
file by using # and/or (#  #).
```

```
}

Domains
{
  color nominal {red, green, blue}
}

Attributes
{
background color
number linear { 0, 1, 2 }
length continuous 0, 200
class nominal {c1, c2}
}

Runs
{
  Run_c1
  {
  Mode = Test             # Don't learn - only testing
  }
  Run_c2
  {
  Mode = Test             # Don't learn - only testing
  }
}

Tests
{
  Test1
    {
      Method = ATest         # Possible Values: ATest, EPIC, EPIC_P, EPIC_RB
      Evaluation_of_selector = Strict    # Possible values: Strict, Flexible
      Evaluation_of_conjunction = Strict # Possible values: Strict,
                                         #   Coverage_ratio,
                                         #   Selectors_ratio, Flexible, Min,
                                         #   Min_w, Prod, Avg, Avg_w
      Evaluation_of_disjunction = Max    # Possible values: Max, Prob_sum,
                                         #   Average, Best_only
      Default_class = Other              # Specity name of run or 'Other'
      Full_report = False
      Tolerance = 0.1
      Threshold = 0.1
    }

Input_hypotheses Run_c1
{
  [class=c1]         # Consequest of the class
    <-- [number=1]   # First rule
    <-- [length>=200][background=red] # Second rule
}

Input_hypotheses Run_c2
{
  [class=c2]  # Consequent of the class
    <-- [number=0]  # Only one rule defined
}

Testing_events
{
blue, 1, 34.6, c1
blue, 0, 9.999, c2
}
```

APPENDIX D AQ21 PARSER

1. Reserved Keywords in AQ21 Input Files

The following is a list of reserved keywords in AQ21 input files that cannot be used to define values of domains and therefore cannot be used in evens. For example the following definition is incorrect:

x4 nominal { robot, blue, clouds, continuous }

because of used keyword *continuous*. In such case it is needed to replace the value continuous by for example value *continuous_* in domain definition and all events. Corrected versions of the definition are presented below:

x4 nominal { robot, blue, clouds, continuous_ }
x4 nominal { robot, blue, clouds, continuous1 }

AQ21 parser is not case sensitive for keywords and for example “advise” is equivalent to “ADVISE” and “AdVise”.

advise
attribute_selection_method
attribute_selection_threshold
attribute_selection_tolerance
attributes
chimerge
common_domains
consequent
continuous
cost
counting_attribute
covered_negatives
covered_positives
cross_validation
dci
define
discr
discretized
domains
episode
epsilon
events
events_percentage
frequency
has_a
ignore_attributes
ignore_events
indexinput_hypotheses
integer
is_a
key
lef
lef_partial_star
lef_ps
lef_sort
lef_star
linear
model
n/a
na
nominal
output_attributes
output_hypotheses
pbm_input
random_seed
ranges
runs
significance
split_events_percentage
str
structured
testing_events
tests
tevent

2. Definition of Values in AQ21 Input Files

All values of attributes need to fall into one of three categories STRING, DOUBLE, or INTEGER.

STRING ::= ([a-zA-Z])+(([a-zA-Z])*([0-9])*([\'\-_~&])*)*

As shown above, STRING values start with a letter that can by followed by sequence of letters, digits, or special signs. For example *red*, *red1*, *re1d*, *RED*, and *red's* are valid STRING values, while *1red*, *123*, and *'red* are invalid.

DOUBLE ::= (-)?[0-9]*"."[0-9]+([eE]([+-])?[0-9]+)?

DOUBLE values start with a digit or "–" (minus) sign, followed by digits, mandatory "." sign, mandatory list of digits and optional exponent. For example *-123.456*, *-0.33*, *123.4*, *.123*, and *-123.456E+7* are valid DOUBLE numbers while *23-34*, *12.34.56*, *twenty*, and *+-34* are invalid.

INTEGER ::= (-)?[0-9]+([eE]([+-])?[0-9]+)?

INTEGER values start with optional "-" (minus) sign followed by mandatory list of digits and optional exponent. For example -123, 0, and 23E+33 are valid INTEGER values while *123.456*, and *123+456* are invalid.

A publication of the *Machine Learning and Inference Laboratory*
School of Computational Sciences
George Mason University
Fairfax, VA 22030-4444 U.S.A.
http://www.mli.gmu.edu

Editor: R. S. Michalski
Assistant Editor: K. A. Kaufman

The *Machine Learning and Inference (MLI) Laboratory Reports* are an official publication of the Machine Learning and Inference Laboratory, which has been published continuously since 1971 by R.S. Michalski's research group (until 1987, while the group was at the University of Illinois, they were called ISG (Intelligent Systems Group) Reports, or were part of the Department of Computer Science Reports).