PARTIAL MEMORY LEARNING SYSTEM AQ-PM: THE METHOD AND USER'S GUIDE

by

M. Maloof
R. S. Michalski

Reports of the Machine Learning and Inference Laboratory, MLI 96-8, George Mason University, Fairfax, VA, 1996.
PARTIAL MEMORY
LEARNING SYSTEM AQ-PM:
The Method and User's Guide

Marcus A. Maloof and Ryszard S. Michalski

MLI 96-20

November 1996
PARTIAL MEMORY
LEARNING SYSTEM AQ-PM:
The Method and User’s Guide

Abstract

This report describes the AQ-PM learning system. AQ-PM is an inductive learning system that
learns from examples distributed over time. It is capable of learning static as well as changing
concepts. AQ-PM takes a partial memory learning approach in which representative examples are
computed from induced concepts and retained for future learning. Representative examples are
those training examples that maximally expand characteristic concept descriptions in the
representation space. The AQ-PM system is an extension to the AQ15c inductive learning system.
In this report, we discuss the partial memory learning method and the AQ-PM learning system.
Examples are given illustrating how to set-up and run AQ-PM on a learning problem.

Key words: Concept learning, learning from examples, learning concepts over time, partial
memory models, concept change

Acknowledgments

The authors thank Ken Kaufman, Eric Bloedorn, and Janusz Wnek for many helpful discussions
about the internals of AQ15c. Thanks to Qi Zhang and Seok Won Lee for reviewing drafts of this
report.

This research was conducted in the Machine Learning and Inference Laboratory at George
Mason University. The Laboratory’s research is supported in part by the Advanced Research
Projects Agency under Grant No. N00014-91-J-1854, administered by the Office of Naval
Research, and Grant No. F49620-92-J-0549, administered by the Air Force Office of Scientific
Research, in part by the Office of Naval Research under Grant No. N00014-91-J-1351, and in part
by the National Science Foundation under Grants No. IRI-9020266 and DMI-9496192.
1 Introduction

AQ-PM is an inductive learning system that learns from examples distributed over time. It is capable of learning static as well as changing concepts. AQ-PM takes a partial memory learning approach in which representative examples are computed from induced concepts and retained for future learning. Representative examples are those training examples that maximally expand characteristic concept descriptions in the representation space. The AQ-PM system is an extension to the AQ15c inductive learning system. In this report, we discuss the partial memory learning method and the AQ-PM learning system. Examples are given illustrating how to set-up and run AQ-PM on a learning problem. This report is intended to be a companion document to the AQ15c method and user’s guide (Wnek et al. 1995) and assumes that the reader is familiar with the concepts therein.

2 The AQ-PM Learning Method

This section discusses the AQ-PM learning method at a high level. For more details, see (Maloof 1996). Referring to Algorithm 1, the basic idea is to maintain a set of representative examples over time, and use these representative examples and past hypotheses for inductive learning, which can be accomplished in either a batch or incremental learning mode. AQ-PM uses a batch learning algorithm to learn from past information. Initially, the system starts with no concepts (step 1) and no representative examples (step 2). The system may possess background knowledge. As an optimization, the system does not learn from new training examples that are already covered by the current set of hypotheses, so a set of misclassified examples is computed (Step 4).

For the first time step, since the system has no concepts or representative examples, all new training examples will be missed and the first training set is equivalent to the first data set. Concepts are learned (Step 6) from the training examples. In Step 7, the induced concepts are used to deductively select the representative examples from the training set. The characteristic concept description is used to compute these examples.

In subsequent learning steps, the set of representative examples is unioned with any misclassified training examples (Step 5) and used with the current set of concepts to learn a new set of concepts (Step 6). The new set of representative examples is then computed (Step 7). This process repeats indefinitely.

Algorithm 1: Partial Memory Incremental Learning

Given data sets Data_t, for t = 1..∞
1. Concepts_0 = ∅
2. Representatives_0 = ∅
3. for t = 1 to ∞ do
4. Missed_t = FindMissedExamples(Concepts_{t-1}, Data_t)
5. TrainingSet_t = Representatives_{t-1} ∪ Missed_t
6. Concepts_t = Learn(TrainingSet_t, Concepts_{t-1})
7. Representatives_t = FindRepresentativeExamples(Concepts_t, TrainingSet_t)
8. end

Representative examples are those examples that maximally expand a concept in the representation space. In other words, they are the training examples that lie on the corners, borders, or surfaces of the hyper-rectangle expressed by the characteristic concept description covering a set of training examples. Algorithm 2 computes the set of representative examples that lie on the borders of a set of characteristic concept descriptions.
Algorithm 2: FindRepresentatives
Given a set of characteristic concept descriptions Concepts, and a set of training examples Train, from which Concepts were induced.

1. Representatives = ∅
2. removeRanges(Concepts)
3. for each rule ∈ Concepts do
4.   for each selector ∈ rule do
5.     newRule = extendRange(selector, rule)
6.     Representatives = Representatives ∪ strictMatch(newRule, Train)
7. end
8. end

As an illustration of the FindRepresentatives algorithm, Figure 1 shows a visualization of two classes of the Iris data set (Fisher 1936) taken from the University of California, Irvine, Machine Learning Archive (Merz and Murphy 1996).

![Figure 1: Visualization of the setosa and versicolor training examples.](image)

Each example of an iris is expressed using four attributes: petal length (pl), petal width (pw), sepal length (sl), and sepal width (sw). The characteristic concept descriptions induced from the training examples in Figure 1 are:

- \[ \text{setosa} \iff [\text{sl} = 0..3] \land [\text{sw} = 0 \lor 2..4] \land [\text{pl} = 0] \land [\text{pw} = 0] \]
- \[ \text{versicolor} \iff [\text{sl} = 1..6] \land [\text{pl} = 1] \land [\text{pw} = 1..2] \]

These concepts are visualized in Figure 2. Note that the training examples are overlaid on the concept descriptions. After the FindRepresentatives algorithm is applied to the concept descriptions and training examples in Figure 2, those training examples that lie on the borders of the concept descriptions are kept as representative examples. Figure 3 shows the corresponding representative examples computed for the two classes. Note that rules are matched to training examples in \( n \)-dimensional space, which is why the following example is judged representative:
versicolor $::= [sl = 3] \& [sw = 0] \& [pl = 1] \& [pw = 2]$

Although the characteristic concept description is used for judging representative examples, other types of descriptions (e.g., discriminant descriptions) can be used for reasoning.

Figure 2: Visualization of the setosa and versicolor concept descriptions with overlain training examples.

Figure 3: Visualization of the setosa and versicolor representative examples.
In addition to mechanisms for finding representative examples, the AQ-PM learning system also has mechanisms for managing, aging, and forgetting representative examples. Inductive support mechanisms keep track of how many times an example has been seen. Rule selection mechanisms allow the user to favor rules that cover frequently seen examples or that cover the most old examples. The parameters for accessing these mechanisms are discussed in Section 3. For a detailed description of these functions, see (Maloof 1996).

3 The User’s Guide

AQ-PM is an extension of the AQ15c inductive learning system (Wnek et al. 1995). This section supplements the AQ15c User’s Guide. The command line syntax for the AQ-PM system is:

```
aqpm.run [-v] < <STDIN> > <STDOUT>
```

The executable takes a file from standard input, learns, and writes the results to standard output. The optional verbose switch (-v) shows intermediate results and aids in debugging and in understanding how the program works. Use this switch with caution since the amount of information can be great.

AQ-PM takes as input, two general file structures. For the first learning step, when there are no representative examples and no concepts, the input file format is the following:

```
<parameters table>
[<criterion tables>]
<variables table>
<events tables>
[<events tables>]
```

After the first learning run, the input files for subsequent learning runs should have the following format:

```
<parameters table>
[<criterion tables>]
<variables table>
<inhypos table>
<representatives table>
<events tables>
[<events tables>]
```

Those tables appearing between two square brackets are optional. The general format of AQ input files and these tables (except for the representatives table) is discussed in the AQ15c User’s Guide (Wnek et al. 1995). The variations that are specific to AQ-PM are discussed in the following sections.

3.1 The parameters Table

In addition to the parameters available in AQ15c, several additional parameters were added to select between learning modes and the partial memory learning mechanisms. Although the parameters table is optional in AQ15c, it is required in AQ-PM. That is, in default mode, AQ-PM is functionally equivalent to AQ15c. The following additional parameters are available in AQ-PM. Note that words appearing in parenthesis are the parameter's default value.
run
In AQ15c, the run parameter is used to conduct multiple learning runs using different parameter settings. For this release of AQ-PM it is not possible to conduct multiple runs using different parameter settings.

learning (batch)
The learning parameter is optional and controls what type of learning will be used. The legal values are:

- `batch`: invokes batch learning. In this mode, AQ-PM is functionally equivalent to AQ15c. It is the default learning mode.
- `partial`: invokes partial memory learning, as described above.

ambig (neg)
In addition to the AQ15c ambig parameters, the recent ambig parameter was implemented in AQ-PM. This parameter handles ambiguous examples, which are those examples that have the same attribute values, but different class labels. The recent ambig parameter selects the newest or most recent training or representative example for learning. The older example is ignored.

echo (pve)
The echo parameter specifies which tables are to be printed. Using the ‘m’ value prints the missed examples table which is a list of the new training examples not covered by the current set of hypotheses.

test (m)
In AQ15c, users may specify multiple flexible matching schemes for testing. For this release, AQ-PM allows only one flexible matching parameter per run.

intersect (borders)
The intersect parameter affects how representative examples are computed. New training examples are flexibly matched against a characteristic concept description. Conceptually, this characteristic concept description is an $n$-dimensional hyper-rectangle, where $n$ is the number of attributes used to form the representation space.

- `corners`: Returns the training examples that lie on the corners of the hyper-rectangle formed by the characteristic concept description.
- `borders`: Returns the training examples that lie on the edges of the hyper-rectangle formed by the characteristic concept description (Algorithm 2).

update (reeval)
The update parameter determines how new representative examples are handled with respect to the old representative examples.

- `reeval`: Old representative examples are re-evaluated with the new training examples to form a new set of representative examples (Algorithm 1).
- `accum`: New training examples are evaluated to form a new set of training examples which are unioned with the old set of representative examples.

useinhyp (yes)
The useinhyp parameter determines whether inhypos are used in the learning process.

- `yes`: Learning is conducted using inhypos represented as training examples, representative examples, and any new training examples that are currently misclassified by the current set of inhypos. The use of training examples derived from inhypos is a rule optimization procedure (Wnek et al. 1996).
- `no`: Learning is conducted using representative examples and any new training examples that are currently misclassified by the current set of inhypos.
aging (off)
The aging parameter determines how representative examples are weighted based on their age. This parameter must be used in conjunction with the criterion table.
off  The ages of representative examples are weighted uniformly or linearly in an increasing or decreasing manner with respect to the past, depending on the criterion table settings.
exp  The ages of representative examples are weighted exponentially in an increasing or decreasing manner with respect to the past, depending on the criterion table settings.
log  The ages of representative examples are weighted logarithmically in an increasing or decreasing manner with respect to the past, depending on the criterion table settings.

forget (off)
The forget parameter determines how representative examples are forgotten.
off  Forgetting is not active. All representative examples are kept.
<\n> Engages a time-based forgetting techniques in which representative examples older than n time steps are forgotten.

repsfactor (1)
The repfactors parameter allows the user to additively weight representative examples. This setting must be used in conjunction with the maxrep and minrep criterion table entries.

time (0)
The time parameter is a required parameter that indicates the time and ranges from 0 to 150. The upper limit is arbitrary and can be extended to the maximum allowable integer on a given computing system. Usually this parameter is not set directly by the user. After the initial learning step, the time parameter is output by the system and initialized at 0. Scripts exist for stripping the parameters table from the output file so it can be given without modification to the next learning step.

Example:
parameters
learning mode ambig trim wts maxstar echo criteria test
partial ic recent gen cpx 10 pv default m

parameters
run learning mode useinhypos intersect update time test
1 partial ic yes borders accum 10 q

3.2 The criterion Table

The criteria table implements a lexicographic evaluation function (LEF) for selecting the most preferred rule. See the AQ15c User’s manual for more information about criterion tables. If a criterion table is to be used in AQ-PM, it should be included in the <filename>.domain file. See Section 5.

Six new rule selection criteria have been implemented in this release of AQ-PM. The first two criteria relate to the number of times a representative has been seen, while the second two relate to the age of the representative examples. The final two relate to the coverage of representative examples
maxseen  maximize the number of frequently seen examples covered by a rule.
minseen  minimize the number of frequently seen examples covered by a rule.
maxold   maximize the number of old examples covered by a rule.
minold: minimize the number of old examples covered by a rule.
maxrep: maximize the number of representative examples covered by a rule.
minrep: minimize the number of representative examples covered by a rule.

The maxold and minold criterion table settings can be used in conjunction with the aging parameter to weight the ages of representative examples to larger or smaller extents. For example, to prefer rules that cover older examples whose ages grow exponentially as time proceeds into the past, use the exp aging parameter and the maxold criterion table. To treat all representative examples the same, regardless of their age, use the off aging parameter and the default criterion table. Finally, to prefer rules that cover newer examples whose ages diminish linearly as time proceeds into the past, set the aging parameter to off and use the minold criterion.

The maxrep and minrep settings are used in conjunction with the repfactor parameter. The repfactor parameter is used to weight representative examples and affects how the cost of the coverage of the representatives examples is computed. The repfactor indicates how much each representative example counts with respect to new training examples. For example, if the repfactor is 2, then one representative example will count twice as much as one training example.

Example:
parameters
learning mode ambig trim wts maxstar echo criteria test
partial ic recent gen cpx 10 pv temporal m

temporal-criteria
criterion tolerance
minold 0.0
maxseen 0.0

3.3 The missed Table

The missed table is only an output table. If the ‘m’ value is included as one of the echo parameters, then this table is produced. It is the list of new training examples that are misclassified by the current set of hypotheses.

Example:
cap-missed
bmax bavg blength bcomp bcl bc2 rlength rwidth rarea rcomp rcl rc2
4 1 0 2 8 4 0 1 0 2 7 5
0 0 3 1 3 1 2 2 8 4 1 0
1 6 2 1 4 1 2 2 5 9 3 2

3.4 The representative Table

The representative table is both an input and output table. The representative table is generated by AQ-PM when partial memory learning is active. The table can also be present in the input file to provide the system with a set of representative examples. It follows the same format as the events and events table, but has a different table name, as shown below.

Example:
cap-representative
bmax bavg blength bcomp bcl bc2 rlength rwidth rarea rcomp rcl rc2
2 2 0 2 8 8 0 1 1 3 7 5
1 1 2 1 4 1 1 2 5 0 3 2
0 0 3 1 0 0 2 2 8 0 0 0

4 Supporting Scripts

In AQ15 (Hong et al. 1986), the Pascal version, incremental learning is completely internalized within the executable. Unfortunately, this internalization of the functions that distribute training data over time, make it difficult run certain experimental designs (such as the STAGGER concept experiments), test against different data sets during a single run, and run other incremental learning methods on exactly the same training data for a given time step. For these and other reasons, the data manipulation functions were not hard coded into the AQ-PM system. Therefore, users must write scripts (Perl, ksh, csh, etc.) to partition data and to assemble AQ-PM input files in the manner they wish. Several scripts are provided with the AQ-PM release to provide a baseline capability.

The high-level Perl script that is responsible for running the AQ-PM system repeatedly over time is run.aqpm, shown in Appendix A. This script calls the AQ-PM executable, aqpm.run, various Unix commands, and three other Perl scripts described below. run.aqpm takes two arguments: the file stem of the input files and the number of data partitions. The data partitions should be of the form <filestem>.data, <filestem>.2.data,..., <filestem>.n-1.data, where n is the number of partitions. The run.aqpm script must be executed in the directory containing the various input files.

The run.aqpm script expects to find a .lparam, .domain, .test, and various .data files. For example, if the filestem were 'caps', then the directory should contain the following files: caps.lparam, caps.domain, caps.test, and caps0.data, caps1.data,..., caps9.data. The caps.lparam file contains the learning parameters for the first learning run. The caps.domain file contains the variables table for all learning runs. The caps.test file typically contains all available training data in a series of event tables, although other testing options are possible. Finally, the various .data files contains event tables for each class and a set of training events for the class. See the AQ15c user's guide (Wnek et al. 1995) for more information about preparing AQ15c data files.

Assuming a directory contained the above files, to start one learning run with AQ-PM, the user would type at the Unix prompt:

% run.aqpm caps 10

This would start the run.aqpm script, which assemble input files, call AQ-PM, disassemble output files, and repeat until all the data partitions have been processed.

The xparam Perl script (Appendix B) takes an AQ-PM file as standard input and writes the parameters table to standard output. Note that this script assumes that the variables table immediately follows the parameter table. Therefore, the 'v' option must be set for the echo parameter. Otherwise, the user can modify the xparam script. The parameters that are stripped from an output file are used as the input learning parameters for the following run. The run.aqpm script saves the parameters in a file named <filestem>.l.lparam, where <filestem> is the first parameter to the script and i is the current partition number. The learning parameters for the 5th run of the caps experiment would be saved to the file named caps5.lparam.

The xhypos Perl script (Appendix C) takes an AQ-PM file as standard input and writes the outhypo as inhypos, stripping off the t- and u-weights. These hypo's are used as inhypos to the next learning step. The run.aqpm script saves the inhypos in the file named <filestem>i.hypos. For the 5th iteration of the caps experiment, the outhypo's would be renamed and saved to the file caps5.hypos. Saving the hypo's in this manner allows for the user to accumulate statistics on the complexity of the rules by writing scripts to count selectors or internal disjuncts.

The xreps Perl script (Appendix D) takes an AQ-PM file as standard input and writes the representative examples tables to standard output. The output representative examples from one learning run are used as the input representative examples in the subsequent run. Again, saving the file in this manner, allows the user to compute statistics on how many representative examples are being stored at each partition.
5 Examples of AQ-PM Learning Runs

5.1 A single run at the beginning of time

When AQ-PM receives a file that only has event and tevent tables (i.e., no representative examples and no inhypos), it is the beginning of time, so to speak. Most learning runs will begin in this manner, although this is not a requirement. The simplest input file to AQ-PM must consist of a parameters table, a variables table, and a set of event tables. Throughout this section we will use the blasting caps data set as an example, because these data files are included with the AQ-PM release. The following is a very simple input file for the caps problem. Assume it is named captest.aqin.

parameters
learning
partial

variables
name  levels  type  cost
bmax  5      lin  1.0
bavg  4      lin  1.0
bleNGTH 6    lin  1.0
bcomp 4      lin  1.0
bc1  10     lin  1.0
bc2  12     lin  1.0
rlegTH 4    lin  1.0
rwidth 5     lin  1.0
rarea 11    lin  1.0
rccomp 6     lin  1.0
rcl  10     lin  1.0
rc2  7      lin  1.0
rsigma 14   lin  1.0
rd  5       lin  1.0
rdl  7      lin  1.0

cap-events
bmax  bavg  blength  bcomp  bc1  bc2  rlength  rwidth  rarea  rccomp  rcl  rc2  rsigma  rd  rdl
0   0   3   1   0   0   2   2   8   0   0   0   7   1   1
1   1   2   1   4   1   1   2   5   0   3   2   2   2   4
2   2   0   2   8   8   0   1   1   3   7   5   3   1   1

non-cap-events
bmax  bavg  blength  bcomp  bc1  bc2  rlength  rwidth  rarea  rccomp  rcl  rc2  rsigma  rd  rdl
1   2   0   2   8   3   0   2   0   2   8   4   2   1   0
2   2   1   1   8   5   1   3   9   1   3   3   2   3   5
1   0   1   1   1   0   0   2   2   2   1   1   0   1   3
1   1   1   1   5   1   0   2   0   4   8   3   0   0   0

Now, we can run AQ-PM by typing:

% aqpm.run < captest.aqin >! captest.aqout

When the command completes execution, the file captest.aqout looks like the following:

parameters
run learning mode anig trim wts nxstar echo criteria verbose useinhypos intersect update time
1 partial ic neg mini cpx 10  pv default 1 yes borders reeval 0

variables
# type  levels  cost  name
1 lin  5  1.00  bmax.bmax
2 lin  4  1.00  bavg.bavg
3 lin  6  1.00  blength.blength
4 lin  4  1.00  bcomp.bcomp
5 lin 10 1.00 crl.bcl  
6 lin 12 1.00 bcl2.bcl2  
7 lin 4 1.00 rlength.rlength  
8 lin 5 1.00 rwidth.rwidth  
9 lin 11 1.00 rarea.rarea  
10 lin 6 1.00 rcomp.rcomp  
11 lin 10 1.00 rcl.rcl  
12 lin 7 1.00 rc2.rc2  
13 lin 14 1.00 rsigna.rsigna  
14 lin 5 1.00 rd.rd  
15 lin 7 1.00 rdl.rdl  

cap-outl hype  
# cpk  
1 [rsigna=2..3,7] [rd=1,4] (t:6, u:6)  

noncap-outl hype  
# cpk  
1 [blength=0..1] [rsigna=0,2] (t:8, u:8)  

cap-representative  
<table>
<thead>
<tr>
<th>bmax</th>
<th>bavg</th>
<th>blength</th>
<th>bcomp</th>
<th>bcl</th>
<th>bcl2</th>
<th>rlength</th>
<th>rwidth</th>
<th>rarea</th>
<th>rcomp</th>
<th>rcl</th>
<th>rc2</th>
<th>rsigna</th>
<th>rd</th>
<th>rdl</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

noncap-representative  
<table>
<thead>
<tr>
<th>bmax</th>
<th>bavg</th>
<th>blength</th>
<th>bcomp</th>
<th>bcl</th>
<th>bcl2</th>
<th>rlength</th>
<th>rwidth</th>
<th>rarea</th>
<th>rcomp</th>
<th>rcl</th>
<th>rc2</th>
<th>rsigna</th>
<th>rd</th>
<th>rdl</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Learning system time: 0.233 seconds  
Learning user time: 1.00 seconds  

If a subsequent learning run is needed, the parameters, outhypos, and representative examples should be stripped from the output file, modified, and used in the input for the next learning step.

5.2 A single run at some point in time

Once we have our first set of representative examples and input hypos, then the anatomy of an AQPM input becomes more complicated. We will need the parameters, variables, and events tables, but we also need the representative and inl hypo tables. Here is an example of a cap1.aqin file:

```
parameters  
run learning mode ambig trim wts mextar echo criteria verbose useinhypos intersect update time  
1 partial ic empty gen cpk 10 pv default 1 yes borders accum 0  

variables  
name levels type cost  
max 5 lin 1.0  
bavg 4 lin 1.0  
blength 6 lin 1.0  
bbcomp 4 lin 1.0  
bcl 10 lin 1.0  
bcl2 12 lin 1.0  
rlength 4 lin 1.0  
rwidth 5 lin 1.0  
rarea 11 lin 1.0  
```
From the command line, we can type:

```
% aqpm.run < capl.acin >! capl.aqout
```

which produces the following output file:

```
parameters
run learning mode ambig trim wts maxstar echo criteria variance useinhypos intersect update time
1 partial ic empty gen qpx 10 pv default 1 yes borders accum 1
variables
# type levels cost name
1 lin 5 1.00 bmmax
2 lin 4 1.00 bavg
3 lin 6 1.00 blength
4 lin 4 1.00 bcomp
5 lin 10 1.00 bcl
6 lin 12 1.00 bcl2
7 lin 4 1.00 rlength
8 lin 5 1.00 rwidth
9 lin 11 1.00 rarea
10 lin 6 1.00 rcomp
11 lin 10 1.00 rcl
12 lin 7 1.00 rc2
13 lin 14 1.00 rsigma
14 lin 5 1.00 rd
```
5.3 Learning over time

Now that we've seen the basics of AQ-PM input files and output files, let's look at how to run AQ-PM progressively over time. The following is a listing of the $AQPMHOME/caps directory:

```
leocardio$ ls
RENAME  caps.test  caps2.data  caps5.data  caps8.data
caps.domain caps0.data  caps3.data  caps6.data  caps9.data
caps.iparam caps1.data  caps4.data  caps7.data
```

The caps.iparam file looks like the following:

```
leocardio$ cat caps.iparam
parameters
run learning mode warmup trim wts maxstar echo criteria verbose useinit hypotheses intersect update forget test
1 partial ic empty gen cpx 10 pv default 1 yes borders accum off 0
```

The first part of the caps.domain looks like the following:

```
leocardio$ head caps.domain
variables
name  levels  type  cost
imax   5  lin  1.0
bavg   4  lin  1.0
blength 6  lin  1.0
bcomp  4  lin  1.0
bc1     8  lin  1.0
bc2     8  lin  1.0
rlength 4  lin  1.0
rwidth  5  lin  1.0
```

There are 10 data files named caps0.data, caps1.data,..., caps9.data. Each data file contains roughly 10% of the data contained in the caps.test file. The first data file, caps0.data looks like the following:

```
15 lin    7 1.00 rdl.rdl
```

```
cap-out hypo
  # cpx
  1 [oz1u=1.9] [rdl=1.4] (t:6, u:6)

noncap-out hypo
  # cpx
  1 [oz2u=2.4] [rcomp=1.5] (t:10, u:10)
```

```
cap-representative
  imax  bavg  blength  bcomp  bc1  bc2  rlength  rwidth  rarea  rcomp  rcl  rco2  rsigna  rdl  rdl
  2     2     0       2      8    8     0       1     1     3     7     5     3     1     1
  1     1     2       1      4    1    1       2     5     0     3     2     2     2     4
  0     0     3       1      0    0     2       2     8     0     0     7     1     1

noncap-representative
  imax  bavg  blength  bcomp  bc1  bc2  rlength  rwidth  rarea  rcomp  rcl  rco2  rsigna  rdl  rdl
  2     2     3       0      5    1    0       3     0     1     8     3     8     0     0
  1     1     1       1      5    1    0       2     0     4     8     3     0     0     0
  1     0     1       1      1    0    0       2     2     2     1     1     0     1     3
  2     2     1       1      8    5    1       3     9     1     3     3     2     3     5
  1     2     0       2      8    3    0       2     0     2     4     2     1     0
```

Learning system time: 0.467 seconds
Learning user time: 0.00 seconds
Finally, the caps.test file contains the testing data. For this problem, the testing file contains all of the original data. So the overall scheme is to learn on 10% of the original data, test on 100% of the original data, learn on the next 10% of the original data, test on 100% of the original data, and so on, until the data partitions have been exhausted. At that point, the learner has seen 100% of the original data. Ideally, testing at this point should produce 100% accuracy. The caps.test file looks like the following:

Once the data files have been set up properly, run the run.aqpm Perl script:

The learning process writes several types of files. The *reps files contain the representative examples. The *.hypos files contain the inhypos for the next learning step. The caps[0-9].lparam files contain the learning parameters for the next learning step. The *.aqpm files are the assembled input files for AQ-PM. The *.aqout files are the output files from AQ-PM.
From these basic files, a great deal of information can be extracted using simple grep commands. Additional information can be extracted using Perl scripts. Predictive accuracy, learning times, testing times, number of selectors, and number of rules can be extracted from the various *.aquot files. The memory requirements for the learning run can be computed by counting the number of representative examples (not lines) in the various *.reps files. The following is an example of how to extract predictive accuracy from the *.aquot files using the Unix grep command.

Leonardo grep Accuracy *.aquot

cape0.aquot: # Events Correct : 45 # Events Incorrect : 21 Accuracy : 68.18%
cape1.aquot: # Events Correct : 60 # Events Incorrect : 6 Accuracy : 90.89%
cape2.aquot: # Events Correct : 53 # Events Incorrect : 13 Accuracy : 80.30%
cape3.aquot: # Events Correct : 58 # Events Incorrect : 8 Accuracy : 87.88%
cape4.aquot: # Events Correct : 58 # Events Incorrect : 8 Accuracy : 87.88%
cape5.aquot: # Events Correct : 63 # Events Incorrect : 3 Accuracy : 95.65%
cape6.aquot: # Events Correct : 62 # Events Incorrect : 4 Accuracy : 93.94%
cape7.aquot: # Events Correct : 59 # Events Incorrect : 7 Accuracy : 95.39%
cape8.aquot: # Events Correct : 59 # Events Incorrect : 7 Accuracy : 95.39%
cape9.aquot: # Events Correct : 65 # Events Incorrect : 1 Accuracy : 98.48%

6 Coding Projects

The following is a list of suggested programming projects that will incorporate additional aspects of the partial memory learning methodology into the AQ-PM learning system. See (Malloof 1996) for more detail. These items are not presented in any order reflecting importance or difficulty.

1. Implement the surfaces intersection method for selecting the representative examples that lie on the surfaces of the n-dimensional hyper-rectangle expressed by a characteristic concept description.
2. Implement frequency-based forgetting in which, after a user-indicated period of time, any example that has not been seen a certain number of time is forgotten.
3. Implement memory-based forgetting in which a certain number of representative examples are kept. If the certain number is exceeded, then some secondary policy (e.g., time-based forgetting) is used to forget examples (e.g., the oldest).
4. Implement time-based rule matching functions in which rules covering the newest or oldest examples are preferred in the event of a conflict (i.e., a tie between multiple rules).
5. Implement frequency-based rule matching functions in which rules covering the most or least frequently seen examples are preferred in the event of a conflict.
6. Implement a parameter in which the system learns from all new training examples, not just those that are missed.
7. Port AQ15 (Hong et al. 1986) no memory and full memory incremental learning from Pascal to C and incorporate into the AQ-PM system. Hooks are present in the aq.c file of the AQ-PM distribution.
8. Implement noise handling mechanisms (Aha et. al. 1991).
10. In the file aqpm.c in the AQ-PM distribution, the functions computeBorders and computeCorners do not remove training examples as they are added to the set of representative examples. Instead, the algorithm repeatedly cycles through the entire set of representative examples until all available complexes have been exhausted. A more efficient algorithm would be to terminate the selection process if all complexes have been exhausted and all available training examples have been added to the representatives examples set. This would involve removing the training example from the list after it is added to the set of representative examples.
7 Known Bugs

The only known bug involves the use of nominal variables. If a nominal variable appears as the first entry in the variables table, when the setup routines read a training example from the events table, they apparently think that the first attribute value (which is nominal) is still part of the attributes list. This occurs even though there is an explicit token in the setup grammar in aqparse.y for an end of line character after the attributes list. If a linear variable appears as the first variable, followed by one or more nominal variables, the program works fine. It is not known why this occurs.

8 Conclusions

The AQ-PM learning system is an extension of the AQ15c inductive learning system and an implementation of a partial memory learner. It is capable of learning a static or changing concept over time. The methodology was briefly discussed and illustrated. The implemented components of the methodology are accessed through various parameter settings, which were discussed at length. Several Perl scripts were presented and discussed for data and experiment management. Examples were given for the blasting caps data set to illustrate the anatomy of AQ-PM input and output files. Finally, the Perl scripts were demonstrated to show how AQ-PM learns over time.
References


Hong, J.; Mozetic, I.; and Michalski, R.S. (1986) AQ15: incremental learning of attribute-based descriptions from examples, the method and user’s guide. Technical Report UIUCDCS-F-86-949. Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL.


Wnek, J.; Kaufman, K.; Bloedorn, E.; and Michalski, R.S. (1995) Selective induction learning system AQ15c: the method and user’s guide. Reports of the Machine Learning and Inference Laboratory, MLI95–4. Machine Learning and Inference Laboratory, Department of Computer Science, George Mason University, Fairfax, VA.

Appendix A: run.aqpm Perl script

#!/usr/local/bin/perl
#

if ($#ARGV != 1) {
    print "Format: run.aqpm <filestem> <n>\n";
} else {
    $i = 1;
    printf "Working on partition $i...\n";
    'cat $ARGV[0].lparam $ARGV[0].domain $ARGV[0]$.i.data $ARGV[0].test > $ARGV[0]$.i.aqin';
    'aqpm.run < $ARGV[0]$.i.aqin > $ARGV[0]$.i.aqout';
    'xparams < $ARGV[0]$.i.aqout > $ARGV[0]$.i.lparam';
    'xhypos < $ARGV[0]$.i.aqout > $ARGV[0]$.i.hypos';
    'xreps < $ARGV[0]$.i.aqout > $ARGV[0]$.i.reps';
    for ($i = 2; $i <= $ARGV[1]; $i++) {
        printf "Working on partition $i...\n";
        $j = $i - 1;
        'cat $ARGV[0]$.j.lparam $ARGV[0].domain $ARGV[0]$.j.hypos $ARGV[0]$.j.reps $ARGV[0]$.i.data $ARGV[0].test > $ARGV[0]$.i.aqin';
        'aqpm.run < $ARGV[0]$.i.aqin > $ARGV[0]$.i.aqout';
        'xparams < $ARGV[0]$.i.aqout > $ARGV[0]$.i.lparam';
        'xhypos < $ARGV[0]$.i.aqout > $ARGV[0]$.i.hypos';
        'xreps < $ARGV[0]$.i.aqout > $ARGV[0]$.i.reps';
    } # for
} # else

Appendix B: xparam Perl script

#!/usr/local/bin/perl
#
# xparams < <filestem>.aqout > <filestem>.param
#
# Takes an aqpm output file and extracts the parameters,
#
$paramSeen = 0;
print "\n";
while (<STDIN>) {
    if (/variables/) {
        exit;
    }
    if (/parameters/) {
        $paramSeen = 1;
    }
    if ($paramSeen == 1) {
        print 
    }
} # while
Appendix C: xhypos Perl script

#!/usr/local/bin/perl
#
#  xhypos < <filestem>.aqout > <filestem>.hypos
#
# Takes an aqpm output file and extracts the outhypos, 
# renames them as inhypos, and strips off the t- and 
# u-weights, and prints them to standard output.
#
$hypoSeen = 0;
print "\n";
while (<STDIN>) {
  if (/^-representative$/) {
    exit;
  }
  if (/^-outhypo$/) {
    $hypoSeen = 1;
    s/outhypo/inhypo/;
  }
  if ($hypoSeen == 1) {
    #s/\(t:.*)\)/;
    print $_; 
  }
} # while
Appendix D: xreps Perl script

#!/usr/local/bin/perl
#
# xreps < <filestem>.agout > <filestem>.reps
#
# Takes an aqpm output file and extracts the representative examples and prints them to standard output.
#
$repsSeen = 0;
print "\n";
while (<STDIN>) {
    if (/Learning system/) {
        exit;
    }
    if (/-representative/) {
        $repsSeen = 1;
    }
    if ($repsSeen == 1) {
        print $;
    }
} # while