85-6

File No. UIUCDCS-F-85-935

85-6

Topics in Incremental Learning of Discriminant Descriptions

Jeffrey M. Becker

**Department of Computer Science
University of Illinois
at Urbana-Champaign**

February 1985

ISG 85-5

This work was supported in part by the National Science Foundation under grant NSF DCR 84-06801

Acknowledgements

This work was supported by the National Science Foundation under grant NSF DCR 84-06801. Thanks go to Professor R. S. Michalski for supervising this project, and contributing ideas and goals, and to the University of Illinois Department of Computer Science for providing the excellent facilities which made this work possible.

Table of Contents

Topics in Incremental Learning
# of Discriminant Descriptions

Abstract

Discriminant descriptions may be learned from classified sets of examples of objects. These descriptions may be used as decision rules. In many situations it is desirable to be able to easily modify existing rules to make them consistent with newly observed examples. This paper covers a number of topics in the area of learning discriminant descriptions from examples. Topics covered include lopsidedness, the N-promise algorithm, the SPLIT algorithm, direct simplification, constructive induction, learning hierarchies of class descriptions, and learning situation-action rules.

1. Introduction

Learning from examples is one of the most common forms of learning. Learning in problem domains in the "real world" is rarely done in batch mode. Generalizations must be formed from a limited number of examples, and gradually modified in the light of new training examples as they become available. It is reasonable to require of learning programs that they be capable of making use of training examples in an incremental fashion.

One type of learning from examples wich is of particular interest is learning *discriminant descriptions.*

"A discriminant description is a description of a class of objects in the context of a *fixed* set of other classes of objects. It states only those properties of the objects in the given class that are necessary to distinguish them from the objects in the other classes" [Dietterich and Michalski, '83a].

Discriminant descriptions may also be viewed in terms of discriminating the objects in one subclass of a given class of objects from the objects in all other subclasses. Thus, hierarchical taxonomies can be specified with discriminant descriptions.

Given a system capable of learning discriminant descriptions, what kinds of problems could it be applied to? It is important to notice that the word "objects" used in the above definition should be considered in the most general sense. It may refer to actual physical objects, conceptual objects, or anything else which may be described in terms of a set of properties such as diseases or situations. Consider the analysis of scientific data. Raw scientific data is usually of little value. Some methods for organizing and summarizing the results must be applied. Numerical statistical methods for analyzing such data have existed for a long time. Recently, techniques have been developed for learning *conceptual* descriptions of classes of objects from examples in the form of logical expressions. Such a system may be used to produce rules for "human consumption" as a statistical analysis tool. For example, we might wish to characterize the difference between cancerous and non-cancerous cells given examples of cells from each class. The description would take the form

features ::> class.

The "::>" symbol is used to distinguish implication linking a concept description with a predicate asserting the concept name from the implication between arbitrary descriptions.

In the case of disease diagnosis, rules would take the form

symptoms ::> cause.

A system has been developed in which such rules are used for "machine consumption" as the rule base for an expert system. A high degree of success has been obtained in the area of plant pathology using machine generated rules for the rule base of an expert diagnosis system [Michalski, Davis, Bisht, and Sinclair, '82]. The system, called Plant/ds, is not capable of incremental rule refinement based on user feedback.

Another possible application, which is untested and is being proposed here, is learning triggering conditions for a finite set of actions. Such rules would take the form

situation ::> action.

The learning system would be coupled with a production system. The rules learned would serve as productions. Such a system would be capable of learning to perform simple tasks from examples. Feedback from the environment or an internal critic function, as well as human input, could be used to provide training instances. For example, we might wish to train a robot to handle the passing of objects from one delivery belt to another and to take certain actions when the belts are not travelling at the same speed. This approach is quite general since actions such as 'deciding an object belongs to a particular class', or 'deciding that a set of symptoms imply the cause is a certain disease with a certain probability' could be implemented as easily as actions involving physical motion.

It should be apparent that the methodology being discussed is capable of providing practical solutions to real world problems. Many other possible applications exist in addition to those mentioned. In the next section, previous approaches to the task of incremental learning are considered. Section 3 provides a discussion of the problem of "lopsidedness" in discriminant descriptions, Section 4 provides an in-depth look at operators which are useful for incremental rule refinement, Section 5 covers issues beyond incremental rule refinement, and Section 6 covers learning of situation-action rules. Appendix A gives a brief summary of the VL1 representation language and related terminology, Appendix B gives the AQ algorithm, and test runs using some current implementations are given in Appendix C.

2. Previous Approaches

Previous approaches to incremental learning have been limited in capability and/or generality. Two systems which may be applied to a wide variety of problems but which do not go beyond incremental rule refinement are AQ11 and ID3. Most other systems do not form discriminant descriptions and are either highly domain dependent or have other serious shortcomings. A few of these other systems will be discussed for comparison.

2.1. AQ11

AQ11 is a program for incremental generation of VL1 hypotheses based upon the AQ algorithm [Michalski and Larson, '83]. For further discussion of the VL1 representation language see Appendix A. For a description of the AQ algorithm see Appendix B. AQ11 operates on VL1 expressions in disjunctive normal form (DVL1 descriptions). The method of inductive generalization used in the AQ algorithm is the *extension against* rule:

Given

CTX_1 & $[L = R_1]$ ::> K {Positive Example}
CTX_2 & $[L = R_2]$ ::> ¬K {Negative Example}

generalize to

$[L \neq R_2]$ ::> K {Output Assertion}

where CTX_1 and CTX_2 are arbitrary expressions (context descriptions) which complete the concept description, and the intersection of R_1 and R_2 is empty. Intuitively, the extension against rule states that we assume an event to be true unless it is known to be false. It provides the most general statement consistent with both the positive and negative examples. Modifications of this rule are used in AQ11 wherein R_2 in the output assertion is replaced by some superset of R_2 which does not intersect with R_1.

As discussed in Appendix B, the AQ algorithm may be used to form the cover of a set of positive examples (eplus) against a set of negative examples (eminus). This is denoted

COVER (eplus | eminus),

read as "cover eplus against eminus." Here, a cover is a DVL1 expression (a union of complexes). An event is covered by such an expression if the event *satisfies* some complex in the expression. If the parts of the two set of examples overlap it is obviously impossible to form a cover. In AQ11, to facilitate the incremental learning process, if the sets of examples overlap the positive examples take priority and

COVER (eplus | eminus) is defined as COVER (eplus | eminus - eplus).

Consider again the definition of extension against given above. Note that the output assertion is defined in terms of R_2. This is important since it means that the negative example is more important for defining the scope of the output assertion than is the positive example. Hence, in forming COVER (eplus | eminus), the set of negative examples is most important in determining the "shape" of the cover produced.

Incremental hypothesis generation in AQ11 proceeds as follows:

Given for each class i, i = 1 .. m, an initial hypothesis H_i and a set of events E_i,

1. Isolate facts which are not consistent with the given hypotheses. For each hypothesis, two set are determined:

   E_i^+ - A set of events which should be covered by H_i but are not.

   E_{ij}^- - A set of events which belong to class j but are covered by H_i and should not be, j $\neq$ i.

2. For each class i determine a cover H_i^- for the exception events E_{ij}^- against the union of all H_i and E_i^+:

$$H_i^- = \bigcup_{\substack{j=1 \\ j \neq i}}^{m} COVER\ (\ E_{ij}^-\ |\ \bigcup_{i=1}^{m} H_i \cup E_i^+\ ).$$

   This step is done because it is computationally more efficient to use formulas H_i^- than the sets of events E_{ij}^-.

3. New hypotheses H_i' are determined as covers:

$$H_i' = COVER\ (\ E_i\ |\ \bigcup_{\substack{k=1 \\ k \neq i}}^{m} H_k - H_k^- \cup E_k^+\ ).$$

A few other details should be mentioned. Complexes in a cover which cover few events may be replaced by the events themselves (for efficiency). If there are few exception events, Step 2 may be bypassed and the union of E_{ij}^- used instead of H_i^- in Step 3.

It has been pointed out by O'Rorke [O'Rorke, '82] that the above procedure may be simplified to:

1. For each class i determine E_i^+, the events which should be covered by hypothesis H_i but are not.

2. New hypotheses H_i' are determined as covers:

$$H_i' = COVER\ (\ E_i\ |\ \bigcup_{\substack{k=1 \\ k \neq i}}^{m} H_k \cup E_k^+\ ).$$

This procedure takes advantage of the fact that

COVER (eplus | eminus) is defined as COVER (eplus | eminus - eplus).

Hence, there is no need to subtract exception events (or a cover of exception events) from the initial hypothesis.

Note that in both procedures the new hypotheses are generated by forming a cover of the events in a class against the union of existing (possibly modified) hypotheses and uncovered events in all other classes. Hypothesis are only used to preserve *negative* information from one hypothesis generation step to the next. The effect of this is *instability*. Instability is observed as a radical change in hypothesis complexity from one generation step to the next which is not warranted by the new training events. AQ11 may exhibit a cyclic instability where covers cycle between being relatively complex and relatively simple, as noted by O'Rorke and others [O'Rorke, '82]. In fact, there is no reason to discard the positive information. Since the COVER function may be applied to formulas as well as events, Step 3 in the first procedure may be replaced by:

$$H_i' = COVER\ (\ H_i - H_i^- \cup E_i^+ \ \mid\ \bigcup_{\substack{k=1 \\ k \neq i}}^{m} H_k - H_k^- \cup E_k^+\ ).$$

Clearly, incremental hypothesis generation consists of two major steps: 1) A specialization step to uncover new exception events. 2) A generalization step to cover new positive events which are uncovered. The specialization step has not been adequately dealt with in the original AQ11 procedure nor in O'Rorkes revision. A new function for selective specialization of complexes, called the SPLIT algorithm, is presented in Section 4, along with a new procedure for incremental generation of VL1 hypotheses.

Coming up with a simple formula for an order of magnitude speed estimate for the AQ algorithm is complicated by the truncation of the partial star list during star formation and the nature of the evaluation functions used to do the truncation. The following rough estimate of computation time for forming a class cover ignores many details of the AQ algorithm and is not guaranteed to proportional to observed times. But, changes to the values of the factors in the following formula should be observable in a corresponding change in computation time:

Formula 2.1.1

(complexity) * (# of attributes) * (MaxStar) * (# of negative examples)

complexity - is measured as the number of complexes in a cover and corresponds to the number of iterations of the AQ algorithm needed to form a class cover.

# of attributes - is the number of attributes used to describe objects. The speed of the EXTENDAGAINST, MULTIPLY, and INCLUDES functions are all roughly proportional to the number of attributes.

MaxStar - is the user supplied limit on partial star size used for partial star truncation. This value is assumed to be roughly the average partial star size.

# of negative examples - is the number of negative examples supplied to the AQ algorithm. The extension of the seed event against each negative example is multiplied into the partial star during star generation.

The least reliable factor in this formula is MaxStar since in actual operation the partial star size may never reach MaxStar or may at times grow much greater than MaxStar. When no truncation takes place, MaxStar should be replaced by (# of negative examples). Hence, the time required by the AQ algorithm could grow by (# of negative examples)^2.

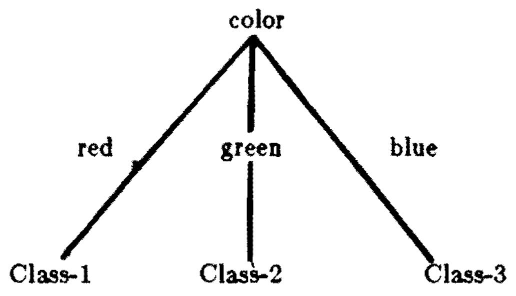
2.2. ID3

ID3 is a program for iteratively constructing decision trees for discriminating between objects of different classes. Objects are represented as feature vectors. Thus, an ID3 decision tree is a kind of discriminant description. ID3 is incremental only in the sense that it takes several tries to find a correct rule (decision tree) when working on large sets of training instances. It builds an entirely new decision tree at each step rather than modifying the existing one.

ID3 decision trees are a more restrictive representation than DVL1 formulas. Each node represents a single object attribute. Each branch from a node represents a single value for the attribute represented by the node.

> "An object is classified by starting at the root of the decision tree, finding the value of the tested attribute in the given object, taking the branch appropriate to that value, and continuing in the same fashion until a leaf is reached. Notice that classifying a particular object may involve evaluating only a small number of attributes depending on the length of the path from the root of the tree to the appropriate leaf" [Quinlan, '83].

The classes to which sets of objects belong are represented at the leaves of the decision tree. This is an example of a one level decision tree:



The basis of the ID3 program is the IP algorithm (for *Iterative Polychotomization).* IP constructs a decision tree in a top-down divide-and-conquer fashion. The outline of the IP algorithm is as follows:

```
IP (Examples, Node)

  If  (Examples = NULL)
      Then Return.
  If  (all Examples belong to the same class)
      Then make a leaf for that class at the current Node and Return.

  Otherwise,

  1. Select an attribute A using some preference criteria.
  2. Assign A to the current node.
  3. For each value V of attribute A
        Sprout a branch and assign V to it,
        Make a new node N,
        Let $Examples_V$ = all Examples with value V for attribute A,
        IP ($Examples_V$, N),
     end.
```

The key to constructing a good decision tree, i.e. one with a small height and few leaves which correspond to empty sets of examples, is picking the best attribute to split on at each node. ID3 uses an information theoretic measure based on relative frequencies of attribute value occurences.

The iterative aspect of ID3 is important when dealing with very large sets of examples. The method is summarized in [Quinlan, '83] as:

```
• Select at random a subset of the given instances (called a window).
• Repeat
      o form a rule to explain the current window
      o find the exceptions to this rule in the remaining instances
      o form a new window from the current window and the exceptions to
          the rule generated from it
  Until there are no exceptions to the rule.
```

According to Quinlan, this method converges rapidly to a correct decision tree, and the final window may contain only a small fraction of the objects being classified.

The cost in terms of computational time of ID3 program "increases only linearly with difficulty as modeled by the product of: the number of exemplary objects, the number of attributes used to describe objects, and the complexity of the concept developed (measured by the number of nodes on the decision tree)" [Quinlan, '83]. This speed is gained at the expense of rule complexity, rule readability, and representational power.

2.3. Other Machine Learning Methods

Many learning techniques are highly domain dependent. Usually, human designed rules of thumb (heuristics) are used to provide the intelligence in such systems. Here are some examples.

2.3.1. Explanatory Schema Acquisition

Explanatory Schema Acquisition is a technique for learning by observation which involves knowledge-based generalization of explanatory schema. The problem domain being studied is story understanding. Explanatory schema are networks of frame-like objects which describe typical story situations. Currently, the types of generalization handled are schema composition, secondary effect elevation, schema alteration, and volitionalization. The generalization types correspond roughly to certain high level reasoning processes in humans. Various heuristics are used to decide whether and how to generalize a schema. The current computer implementation only handles one special type of story [DeJong, '83].

2.3.2. Learning by Augmenting Rules and Accumulating Censors

This technique is based on learning from precedents by drawing analogies. Because rules learned by analogy are not usually totally correct they must be "fixed up". Augmented rules have an "unless" part, in addition to the usual "if" and "then" parts, which prevents the "if" part from being satisfied under certain conditions. Censors are rules which may trigger the "unless" part of other rules. Again, the problem domain being studied is story understanding. This methodology seems fairly general, except for the analogical matching process which appears to be tailored to the problem domain. Semantic nets are used to represent story situations. A corresponding predicate logic representation is used for representing if-then-unless rules learned by the system [Winston, '83b].

2.3.3. The Instructible Production System Project

The Instructible Production System Project produced a series of systems for learning from instruction. Rules for driving the learning process were encoded in a kernel of productions along with rules for driving the user interface and rules for performing tasks in the task environment. The starting size of the kernels ranged from 295 to 450 rules. The systems were designed for goal oriented problem solving with feedback and instruction being supplied by interaction with a human instructor. Various rules were developed for reconstructing rules from existing rules. Although much may have been learned by the design effort, all of the systems fell short of Rychner's hopes in terms of flexibility and robustness [Rychner, '83].

2.3.4. Learning by Discovery

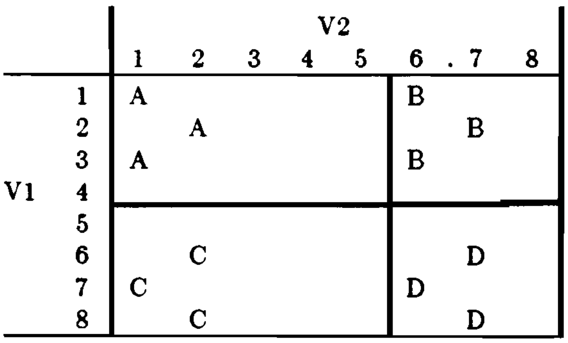
AM and EURISKO are heuristically driven programs for learning by discovery. As with the previous knowledge based systems, these systems start with an initial kernel of knowledge (in the form of heuristics and concepts in this case) and attempt to add to that knowledge. The AM system, designed for discovering concepts in the field of mathematics, started with 115 core concepts and 243 rules for driving the heuristic search for plausible new concepts. The EURISKO system was designed to discover new and useful heuristics using heuristics. Both programs have achieved promising results, especially in the area of constructive induction [Lenat, '83].

2.4. Conclusions

AQ11 and ID3 are general purpose systems for learning discriminant descriptions from examples. Most other systems are supplied by the human designer with a kernel of rules which enable them to construct new rules (i.e. learn) by making modifications to existing rules. In general, practitioners of machine learning are working to eliminate the expert knowledge acquisition bottleneck. At this point, it does not appear to be any easier to develop heuristics for guiding knowledge acquisition in a new domain than it would be to encode the expert knowledge directly. The bottleneck has just been shifted up a level. A worthwhile goal would be to build a system with basic learning capabilities which could learn rules for efficient rule construction in new domains, i.e. a system which "learns how to learn". This is an important capability which people seem to possess. Although at this point the basis of such a system is open to speculation, a good starting point appears to be learning situation-action rules as discriminant descriptions from examples.

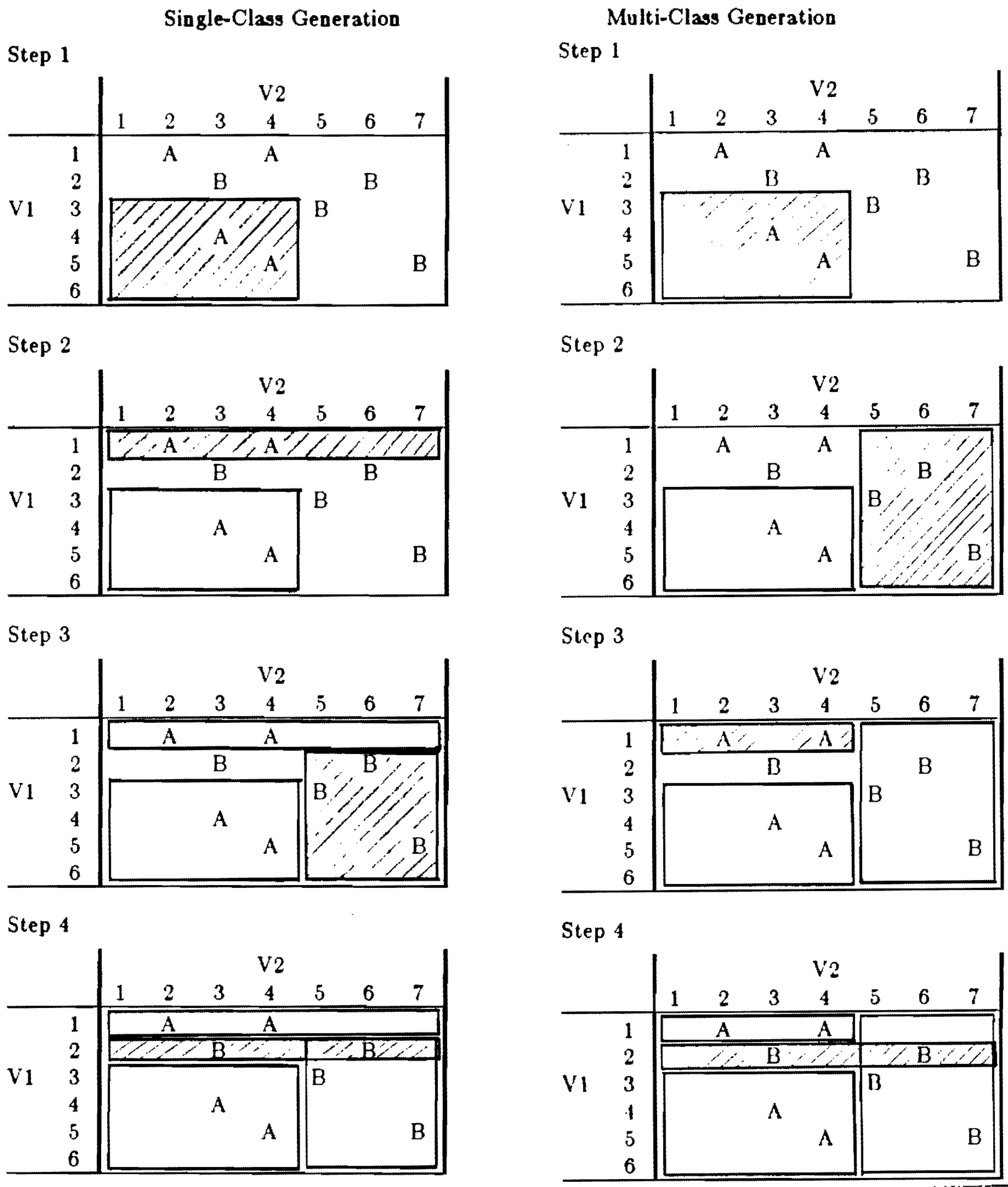
3. An Inherent Problem: Lopsidedness

There is an inherent problem in the formation of discriminant descriptions in a finite representation space which I will refer to as "lopsidedness." Lopsidedness is observed as an increase in complexity of concept descriptions (or covers) from those formed early to those formed later, even though the underlying concepts are equally complex. Consider the following diagram, where the outer box represents the event space boundary, the lines represent a partitioning of the event space, and letters represent examples in a class. Assume V1 and V2 have NOMINAL domains with the indicated value sets.



This illustrates the typical distribution of unobserved events in an event space in covers formed by a discriminant description learning system if class covers are formed in the order A first, B second, C third, and D fourth. Here, class covers are required to be disjoint. Covers formed early "consume" large volumes of the event space (as shown for Class A in the diagram). Those covers formed later may not overlap with existing covers, hence they must be "smaller", which means more specialized and more complex.

When the AQ algorithm is being used to form disjoint discriminant descriptions, both lopsidedness and speed may be improved upon somewhat by using a re-entrant version of AQ which allows forming one covering complex at a time for each class. The top-level algorithm cycles through the classes forming one cover per class until a complete cover has been generated for each class. I call this "multi-class" generation since the cover generation process is active for all classes at once. The diagrams below give a comparative illustration of the single-class generation process versus the multi-class generation process for a simple problem:



Lopsidedness is decreased because the covers are not generated class by class, but rather one covering complex per class at a time, giving each class a better chance of "claiming territory" in the event space. Note that this method will not be very effective at reducing lopsidedness if the number of complexes in the class covers is few. In both cases, once a cover has been generated, it is used in place of the events it covers for further cover generation. Covers are formed using

COVER (eplus | uncovered-eminus-events $\cup$ covers-of-eminus-events).

This effectively decreases the number of negative events (last factor in Formula 2.1.1, Section 2.1). In the multi-class generation case, the effective number of negative events is reduced sooner since we move on to the next class after finding a single covering complex rather than an entire class cover. A typical speed-up of about 1.4 is obtained when using multi-class generation rather than single class generation in the disjoint covers mode. Multi-class generation capabilities are implemented in AQINTERLISP. See Appendix C for a specific example.

Another way to handle the problem of lopsidedness is to control the "degree of generalization". An effective way of measuring the degree of generalization of a cover is by counting the number of unobserved events in the event space which are covered. This measure is called *absolute sparseness.* The *relative* sparseness of a cover is defined as the ratio of absolute sparseness of the cover to the total number of events covered. [Michalski and Stepp, '83b]. By selectively specializing a cover to reduce the relative sparseness to below some threshold value, it would be possible to generate roughly "balanced" rather than lopsided covers. A *trimming* function which reduces the reference of each selector in a complex by finding the intersection of the complex with the refunion of the associated covered events is one way to accomplish this.

If the application in which the rules are to be used allows it, perhaps the best way to deal with lopsidedness is to allow class covers to intersect. Since we are assuming that our learning system has incremental hypothesis generation capabilities, when a new training or testing instance is covered by more than one class cover, the class covers can be appropriately modified. In this case we are allowing some of the don't-care space, or perhaps more appropriately "don't-know" space, to remain ambiguous until adequate information has been recieved by the system for resolving the ambiguity. Such a system might guide the selection of further training instances by requesting or selecting training instances which fall within the scope of more than one class cover, hence speeding the process of resolving the ambiguities.

4. Operators for Incremental Hypothesis Generation

A large part of my current study involves identifying and inventing operators which would be useful for incremental learning of discriminant descriptions. These operators should be top-down whenever possible so that the decision making process is comprehensible and justifiable to the average human. They should be capable of forming rules which are concise and seem reasonable to human experts. These operators should also be reasonably efficient for very large bodies of rules on single processor computers, and should be composed of operations which may be performed in parallel on the highly parallel machines of the future. Two basic steps are needed to do incremental hypothesis generation: 1) A generalization step that modifies a cover to include new positive examples that were not previously covered, and 2) A specialization step that modifies a cover so that it does not cover new negative examples that were previously covered. The steps may be performed in either order. In AQ11 the implicit specialization step is to reduce a cover to the individual events covered and the generalization step is to form a new cover using the AQ algorithm.

The AQ algorithm is not top-down but is acceptably fast on moderately sized problems, is composed of operations many of which may be performed in parallel, and produces very good rules. Hence, it serves very well as a generalization operator. There exists a need for a specialization operator which specializes rules while maintaining conciseness as much as possible. It is desirable to have an operator for identifying the most promising attributes for forming discrimination rules, and for identifying promising sets of attributes for constructive induction. It is also desirable to be able to directly simplify class descriptions since the generalization and specialization operators are not likely to be perfect and may introduce redundant terms.

The next three sections address these needs. Section 4.1 describes the N-promise algorithm for selecting most relevant attributes. In Section 4.2 the SPLIT algorithm, a top-down learning algorithm which uses N-promise and reducing the reference of selectors to learn discriminant descriptions, is described. In Section 4.3 some elementary simplification operations are described.

4.1. The revised PROMISE algorithm: N-promise

In this section a revised version of the promise algorithm for selecting most relevant attributes is described and illustrated. The PROMISE algorithm described in [Baim, 82] is a powerful tool for determining the most relevant attributes for inductive learning systems. However, there is room for making some improvements. First, the value calculated is not normalized so, unlike a probablity value, it is impossible to determine the "meaning" of a particular promise value out of context. Second, a promise value can be computed more directly by summing the appropriate values without the necessity of building relational tables, locating and removing events, etc. Third, the algorithm given will not work for formulas. This new version, called N-promise, has the following features:

1) The computed value is normalized on a scale of 0 to 1 (hence the name N-promise for "normalized promise"). An N-promise value of 0 for an attribute corresponds to the worst case, when an attribute has the same values in all events. An N-promise of 1 corresponds to the best case, when there are no CCEs for an attribute.
2) By using relative frequencies rather than counts and class sizes to determine which values to sum, a more reliable metric is obtained.
3) The algorithm works for formulas (a cover for many points in an event space) as well as for events (single points in an event space).

4.1.1. Comparison of N-promise to PROMISE

This new algorithm has the same limitations and is based on the same assumptions as the original. The original version used relational data base techniques, whereas the new version works more directly from relative frequency tallies. The abbreviations CCE for cross-class equivalency and ICE for in-class equivalency are used throughout. The original algorithm is stated here for reference:

Algorithm PROMISE: This algorithm determines the promise of an attribute by computing a "cost" based on the loss of class-distinguishing information due to CCEs within the projection of the data-set onto the current attribute of interest. It does this by selectively removing events from CCEs and re-evaluating the data-set until no CCEs remain. The algorithm returns a value, p, which is a measure of the promise of the attribute where p ranges from 0 for a perfect attribute (without CCEs) to m-1 for the worst case (when the attribute has the same value in all events) where m is the number of distinct classes in the data-set. The initial value of p is 0.

```
1.  Project the data-set on the attribute.
    Repeat steps 2 through 5 until no more CCEs are found.
2.  Locate an event, e, that is a member of a CCE.
3.  Find the largest class, c, with the smallest ICE within the CCE which contains the
    event found in step 2.
4.  Increase p by the size of the ICE in the class found in step 3 divided by the size of the
    class.
5.  Remove the events in the ICE accounted for in step 4 from class c.
6.  Exit.
```

Let us examine the value that this algorithm computes. Let relative frequency R(a,v,c) be the number of times a value for a particular attribute occurs in the events of a particular class divided by the class size (i.e. the number of events in the class), where the subscripts or indices indicate: a - the attribute, v - the value of attribute a, and c - the class.

For each value of an attribute the algorithm computes:

$$S(a,v) = \sum_c R(a,v,c) - MAX_c\ R(a,v,c)$$

Where

$$MAX_c\ R(a,v,c) = MAX\ (R(a,v,1),\ R(a,v,2),\ \cdots\ ,\ R(a,v,n))$$

for classes numbered 1 .. n.

This is not quite true. Repeated application of step 3 will leave the largest ICE for the smallest class c untouched by steps 4 and 5. This may only approximate MAX_c R(a,v,c), depending on the order of comparison. This problem is corrected in the new algorithm since relative frequencies are used directly.

Let p be the promise value as described by Baim. Study of the above algorithm will show that for a given attribute a,

$$p = \sum_v S(a,v).$$

If m is the number of classes in the current problem then p may have a value between 0 (best possible promise - no CCEs) and m-1 (worst possible promise - attribute is completely redundant). Note that

$$\sum_v \sum_c R(a,v,c) = m.$$

Let p' be promise normalized on a scale of 0 (worst) to 1 (best). Then

$$p' = (m-p-1)/(m-1)$$

$$= \frac{\sum_v \sum_c R(a,v,c) - \sum_v S(a,v) - 1}{m-1}$$

$$= \frac{\sum_v \sum_c R(a,v,c) - \left[\sum_v \sum_c R(a,v,c) - \sum_v MAX_c\ R(a,v,c)\right] - 1}{m-1}$$

$$= \frac{\sum_v MAX_c\ R(a,v,c) - 1}{m-1}$$

Thus it is easy to see that the normalized promise of an attribute may be calculated from the sum of the maximum relative frequencies of its values. This is essentially what the new algorithm does. One major difference is how "class size" is determined. If formulas are allowed then a single complex can contribute more than once to the occurences of values tallied for an attribute. Thus, the number of occurences tallied must be totalled separately for each attribute in each class. This total is then used as the "class size" for that attribute and class. An additional modification is needed to account for the loss of independence between the occurences tallied for the values of an attribute when formulas are allowed. Let C be the class size determined as indicated above, and N be the number of complexes in the class. When computing relative frequencies the weighting factor used is N / C^2 . This reduces to $1 / C$ when all of the complexes in the class are events. As with PROMISE, attributes may be grouped to form *compound* attributes so that interrelationships between attributes may be detected.

This is the N-promise algorithm:

Step 1: Select an attribute "a". Tally the occurences of values for the selected attribute and compute the class size for the attribute. All quantities are initialized to 0. T is the tally of occurences. N is the "class size".

```
for each value v
  for each class c
     T(a,v,c) := {the number of occurences of [a = v] in class c events}
     C(a,c) := C(a,c) + T(a,v,c)
  end
end
```

Step 2: Compute the sum of maximum relative frequencies and normalize. P is the N-promise value (initially 0), N(c) is the number of complexes in class c, w is the weighting factor, and m is the number of classes in the current data set.

```
for each class c
   w := $N(c)/C(a,c)^2$
   for each value v
      P := P + $MAX_c$ (T(a,v,c) * w)
   end
end
P := (P-1)/(m-1)          (* normalize *)
```

4.1.2. N-promise Example

An example will illustrate the functioning of the algorithm. The tables and accompanying descriptions show a sample data set and the results of the computations performed by the N-promise algorithm. The first table below gives the domain descriptions of the variables (attributes) being used. The second table gives complexes which are examples of three classes, where each row represents a complex. The second complex in class A is a formula, the remainder are events.

DATA SET

Domain Descriptions		
Variable(s)	Domain Type	Values
V1,V3	Nominal	1,2
V2	Nominal	1,2,3
V4	Nominal	1,2,3,4

Events and Formulas				
Class	V1	V2	V3	V4
A	1	2	2	3
	1	1,2,3	1	2,3
B	1	1	1	1
	1	2	2	1
	2	2	2	1
C	2	1	1	3
	2	1	2	3
	2	3	1	2

COMPUTATIONS

The first step is to tally occurences of variable (attribute) values in the input data and find the class sizes. The first table below shows the occurence tallies for each value of a variable in each class, the second gives the class sizes by class and variable. Note that when no formulas are present, the class size is always equal to the number of events in that class.

Occurence Tallies											
Class	V1		V2			V3		V4			
	1	2	1	2	3	1	2	1	2	3	4
A	2	0	1	2	1	1	1	0	1	2	0
B	2	1	1	2	0	1	2	3	0	0	0
C	0	3	2	0	1	2	1	0	1	2	0

Class Sizes				
Class	V1	V2	V3	V4
A	2	4	2	3
B	3	3	3	3
C	3	3	3	3

In the second step, we divide each occurence count by the associated class size to get the following relative frequencies, with maximums indicated at the bottom of each column. These maximums are then summed to get a "raw" value which is then normalized, yielding an N-promise value for each variable.

Relative Frequencies											
Class	V1		V2			V3		V4			
	1	2	1	2	3	1	2	1	2	3	4
A	1	0	1/8	1/4	1/8	1/2	1/2	0	2/9	4/9	0
B	2/3	1/3	1/3	2/3	0	1/3	2/3	1	0	0	0
C	0	1	2/3	0	1/3	2/3	1/3	0	1/3	2/3	0
max	1	1	2/3	2/3	1/3	2/3	2/3	1	1/3	2/3	0

N-promise values				
	V1	V2	V3	V4
Raw	2	5/3	4/3	2
Normalized	1/2	1/3	1/6	1/2

From the normalized promise values it is easy to see that V1 and V4 are the most promising attributes, but that neither one alone could be used to completely discriminate the three classes. AQINTERLISP was run on the above data (without the benefit of the N-promise results) to confirm this conclusion. These covers were generated in the non-intersecting covers mode:

Cover of Class: A	$[V1 = 1][V4 \neq 1]$
Cover of Class: B	$[V4 = 1]$
Cover of Class: C	$[V1 = 2][V4 \neq 1]$

4.1.3. Applications of N-promise

N-promise is applicable in all situations that PROMISE is, and possibly others. As with PROMISE, N-promise may be used to evaluate groups of attributes to test for strong interrelationships which could serve to guide the constructive induction process. When background rules are used for constructive induction, as in INDUCE-2, a large number of selectors containing derived attributes could be added to a complex. N-promise results can be used as criteria when trimming a list of background rules to select the rules that are most likely to be productive. N-promise results can also be used to discover and remove redundant selectors prior to applying inductive-learning techniques, whether or not selectors containing derived attributes have been added.

Because it works for formulas, and because of the organization of the computation, N-promise can be used efficiently for selecting most promising attributes when using the SPLIT algorithm. This methodology is described in detail in the next section. N-promise could also be used for attribute selection in programs such as ID3 which build decision trees.

Similarity may be defined in a number of ways. One way, proposed by Tversky "states that two concepts become more similar as the number of properties shared by them increases and the number of distinctive (i.e. nonshared) properties decreases" [Barslou, '83]. The N-promise algorithm may be used to derive a clustering quality measure for a given clustering of events by summing the N-promise values for the set of attributes (variables) in the current problem. The explanation is quite simple. If the total of the normalized promise values is relatively high, that means the number of ICE's (shared properties) is large for a large proportion of the attributes in each class and the number of CCEs is low. Hence, there must be strong similarities between members of a class.

4.1.4. Conclusions

A revision of the PROMISE algorithm has been introduced. The relationship between the original PROMISE algorithm and N-promise has been shown. The N-promise algorithm has been given, and illustrated with a simple example. The new algorithm is straightforward and easy to understand. It has features which make it useful as an attribute selector for the SPLIT algorithm, and in a number of other applications.

Postscript: In [Baim, 83] Baim describes promise as a normalized value between 0 and 1 exactly as given here, but derived using a slightly more complex formula. The formula given by Baim produces exactly the same value as the one given here.

4.2. The SPLIT Algorithm

In incremental learning it is frequently necessary to specialize overly general class descriptions so that new negative training examples are not covered. It is desirable that this operation be efficient for large sets of rules, and that it maintain the conciseness of the class description being specialized as much as possible. It is also desirable that an AI system performing incremental learning be able to inform the user about why a particular rule was formulated in a particular way. If the rules are to be used in an expert advice system, knowledge about attribute importance can be used to guide the order of questioning. It is especially important that the rules themselves be understandable so that they may be evaluated by a human expert.

As previously discussed, the AQ11 algorithm is efficient for small sets of rules, but becomes more expensive very rapidly as the set of rules grows in size. AQ11 does not actually modify a class description, rather it remakes a new cover of the events in a class against the union of uncovered positive events in the other class and the difference of the existing covers of the other classes and the corresponding covers of exception events [Michalski and Larson, '83]. The Star methodolgy used in AQ11 is a generate and test scheme. So, certain information such as values returned by evaluation criteria functions can be supplied to the user, but this information may not always be adequate. Rules produced by AQ11 are in the VL1 representation language and are easily understood.

The ID3 algorithm forms decision trees in a fast, but rather simple minded way. Information concerning what the system considered the most important attributes can be extracted from the decision tree building process by observing which attributes are selected the earliest. Decision trees produced by ID3 are generally more complex than VL1 rules produced by AQ11 for a given problem [O'Rorke, '82], and they are not easily understood by people.

The SPLIT algorithm, although not concieved as such, has features which make it appear to be a hybrid of AQ11 and ID3 techniques. It uses a top-down divide-and-conquer control scheme remniscent of ID3, so a great deal of information can be extracted from the decision making process during rule formation or modification. It is reasonably efficient, many of the operations could be performed in parallel, and it operates on VL1 expressions so the rules are understandable. An inductive learning algorithm with incremental VL1 hypothesis generation capabilities using SPLIT and AQ is described in part 3 of this section.

4.2.1. Description of the SPLIT algorithm

The SPLIT algorithm accepts as parameters an initial hypothesis (Cover), which is assumed to be a single complex, and lists of positive (Pos) and negative (Neg) training examples covered by the initial hypothesis. The examples may be events or formulas. It specializes the hypothesis by recursively "splitting on a selector" until none of the negative examples are covered. This is done in such a way that the simplicity of the resulting hypothesis is preserved as much as possible and all positive examples remain covered. Splitting on a selector involves separating the values of the right hand side (RHS) of the selector into two groups to allow changing a present internal disjunction into an external disjunction. For example,

$$[X = 1][Y = 2][Z = a,b,c]$$

might be split to form:

$$[X = 1][Y = 2][Z = a] \vee [X = 1][Y = 2][Z = b,c].$$

If one of the complexes resulting from a split still covers both positive and negative examples, it is split further. If it covers only positive examples it is left as-is, and if it covers only negative examples it is deleted. Special criteria are used to decide which selector to split on, and how to distribute values from the RHS of the selector being split on.

This is the SPLIT algorithm:

```
SPLIT (Cover, Pos, Neg)

    If Neg = NULL then Return Cover
    If Pos = NULL then Return NULL

    Otherwise do:

    1. Pick PSel, the selector containing the most promising attribute in Cover,
       using N-promise and tie-breaking heuristics.

    2. Divide the values in the RHS of PSel into two groups,
       Vals+ for those with the most occurences in Pos examples, and
       Vals- for those with the most occurences in Neg examples.
       For ties, assign a value from the RHS PSel to either Vals+ or Vals-
       according to tie-breaking heuristics.

    3. Use the division of values from step 2 to transform Cover into the
       disjunction of Cover+ and Cover-.

    4. Divide the complexes in each of Pos and Neg into two groups,
       Pos+ and Neg+ for those containing an occurence of a value in Vals+
                  in the corresponding selector, and
       Pos- and Neg- for those containing an occurence of a value in Vals-
                  in the corresponding selector.
       Note that formulas may end up in BOTH + and - groups.

    5. Return The-Disjunction-of ( SPLIT (Cover+ , Pos+ , Neg+ ),
                                   SPLIT (Cover-, Pos-, Neg-) ).
```

The N-promise algorithm mentioned in step 1 is a revision of the PROMISE algorithm [Baim, 82] and is descibed fully in the preceding section. More should be said about the sets of tie-breaking heuristics mentioned in steps 1 and 2.

For step 1, the following heuristics are recommended:

1) Pick the selector with the fewest values (there must be at least 2) on its RHS. This effectively minimizes the depth of recursion, and so increases in rule complexity are minimized.

2) Don't pick a dropped (ie. [Var = *]) selector. Splitting a dropped selector causes it to be re-introduced into the rule, increasing the rule complexity.

For step 2, the following heuristics are recommended:

1) If one value remains to be assigned and one group has no values assigned, assign the value to the group with no assigned values. Putting all of the values in one group would defeat the purpose of this operation.

2) If the selector has a NOMINAL domain variable - make the split as "balanced" as possible, ie. the value should go to the side with the least number of values assigned.

3) If the selector has a LINEAR domain variable - keep adjacent values together. For example, if for a value Val(i) both Val(i-1) and Val(i+ 1) have been assigned to Vals+ , then Val(i) should be assigned to Vals+ to complete the interval.

4) If the selector has a STRUCTURED domain variable - try to fill in all children under the highest possible parent node.

Given appropriate evaluation functions these heuristics may be implemented using a

"Lexicographical Evaluation Function with tolerances" (LEF) [Michalski and Stepp, 1983]. A LEF is a sequence of criterion-tolerance pairs $(C_1,T_1),(C_2,T_2)$, ..., where C_i is an elementary criterion and T_i is a tolerance threshold ($0 \leq T_i \leq 100\%$). Each criterion C_i is applied in order to a list of items and those items that score within threshold T_i are retained. The process stops when either a single "best" item remains, or the sequence of criterion-tolerance pairs is exhausted. In the latter case, an item is chosen arbitrarily from those remaining.

4.2.2. An Example Using SPLIT

A simple example will demonstrate the operation of the SPLIT algorithm. We will use the Data Set described in the section on N-promise. Let the initial hypothesis be the entire event space, let the positive examples be the complexes in Class A, and let the negative examples be the complexes in Classes B and C. We have,

```
Cover = [V1 = *][V2 = *][V3 = *][V4 = *]
Pos   = {[V1 = 1][V2 = 2][V3 = 2][V4 = 3]
         [V1 = 1][V2 = *][V3 = 1][V4 = 2,3]}
Neg   = {[V1 = 1][V2 = 1][V3 = 1][V4 = 1]
         [V1 = 1][V2 = 2][V3 = 2][V4 = 1]
         [V1 = 2][V2 = 2][V3 = 2][V4 = 1]
         [V1 = 2][V2 = 1][V3 = 1][V4 = 3]
         [V1 = 2][V2 = 1][V3 = 2][V4 = 3]
         [V1 = 2][V2 = 3][V3 = 1][V4 = 2]}
```

In Step 1, N-promise is used to select the most promising attribute. For the attributes in classes Pos and Neg above N-promise returns:

Attribute	N-promise
V1	2/3
V2	0
V3	0
V4	1/3

V1 is selected as the most promising attribute, so PSel is [V1 = *]. In Step 2, the values in PSel are split into two groups. Note that [V1 = *] is equivalent to [V1 = 1,2] since V1 has a Nominal domain consisting of the values 1 and 2. The following table gives the occurence counts for the values in the RHS of PSel for classes Pos and Neg.

Occurence Counts		
Class	V1	
	1	2
Pos	2	0
Neg	2	4

Vals- becomes "2". Vals+ becomes "1" by virtue of the first tie-breaking heuristic. In Step 3 we have:

```
Cover+ = [V1 = 1][V2 = *][V3 = *][V4 = *],
Cover- = [V1 = 2][V2 = *][V3 = *][V4 = *].
```

In Step 4,

```
Pos+  = {[V1 = 1][V2 = 2][V3 = 2][V4 = 3]
         [V1 = 1][V2 = *][V3 = 1][V4 = 2,3]}
Neg+  = {[V1 = 1][V2 = 1][V3 = 1][V4 = 1]
         [V1 = 1][V2 = 2][V3 = 2][V4 = 1]}

Pos-  = NULL
Neg-  = {[V1 = 2][V2 = 2][V3 = 2][V4 = 1]
         [V1 = 2][V2 = 1][V3 = 1][V4 = 3]
         [V1 = 2][V2 = 1][V3 = 2][V4 = 3]
         [V1 = 2][V2 = 3][V3 = 1][V4 = 2]}
```

In Step 5, on the recursive call SPLIT(Cover+ , Pos+ , Neg+), Cover+ must be split further. However, Cover- is deleted (NULL is returned) since it covers no positive examples. Following the recursive call on Cover+ , we now have Cover = Cover+ , Pos = Pos+ , and Neg = Neg+ . In Step 1 the following N-promise values are obtained:

Attribute	N-promise
V1	0
V2	1/8
V3	0
V4	2/3

V4 is selected as the most promising attribute, so PSel is [V4 = *]. The subsequent steps result in:

```
Cover+  = [V1 = 1][V2 = *][V3 = *][V4 = 2,3],
Cover-  = [V1 = 2][V2 = *][V3 = *][V4 = 1,4].

Pos+  = {[V1 = 1][V2 = 2][V3 = 2][V4 = 3]
         [V1 = 1][V2 = *][V3 = 1][V4 = 2,3]}
Neg+  = NULL

Pos-  = NULL
Neg-  = {[V1 = 1][V2 = 1][V3 = 1][V4 = 1]
         [V1 = 1][V2 = 2][V3 = 2][V4 = 1]}
```

On the next recursive call to SPLIT, Cover+ is returned as-is, and Cover- is deleted. The final value for the cover of Class A with [V = *] selectors dropped is [V1 = 1][V4 = 2,3].

The covers of class B and C may be generated in a similar fashion. As a final result we have:

Cover of Class A:	[V1 = 1][V4 = 2,3]
Cover of Class B:	[V4 = 1]
Cover of Class C:	[V1 = 2][V2 $\neq$ 2]

Note that these covers have a non-null intersection. Non-intersecting covers may also be generated. All that is needed to accomplish this is to replace the examples in a class by the new hypothesis (cover) returned by SPLIT before proceeding to the next class.

4.2.3. Incremental Generation of VL1 Hypotheses

We wish to incrementally learn discriminant descriptions for a set of classes by starting out with a set of hypothesized class covers and gradually modifying these in the light of new training instances. Class covers are assumed to be in disjunctive normal form (DNF), i.e.

[complex v complex v ... v complex].

When the SPLIT algorithm is combined with the AQ algorithm and a small number of other functions, we have an effective method for incremental generation of VL1 hypotheses. In this version, SPLIT is used to specialize the existing cover of each class to uncover new negative events which are covered. AQ is then applied to generalize the covers of each class to cover new positive events which are uncovered:

```
for each class
  for each complex $C_i$ in the class-cover
     Pos := {All examples covered by $C_i$, both new and old}
     Neg := {New examples from all other classes
              which have a non-null intersection with $C_i$}
     $C_i$ := SPLIT($C_i$, Pos, Neg)
  end
end
for each class c
   Pos := {Cover(c) $\cup$ uncovered-positive-events}
   Neg := {covers-of-all-other-classes $\cup$ uncovered-negative-events}
   Cover(c) := COVER (Pos | Neg)
end
```

"COVER" uses the AQ algorithm to form a cover of the complexes (formulas or events) in Pos against the complexes in Neg.

4.2.4. Conclusions

The SPLIT algorithm has been introduced and described. It has been demonstrated with a simple example. An algorithm for incremental generation of VL1 hypothesis using the SPLIT algorithm and the AQ algorithm has been described. The cost in terms of computation time of the SPLIT algorithm should proportional to that of the ID3 algorithm, which "increases only linearly with difficulty as modeled by the product of: the number of given exemplary objects, the number of attributes use to describe objects, and the complexity of the concept developed (measured by the number of nodes on the decision tree)" [Quinlan, 83]. The SPLIT algorithm uses more specialized decision criteria and a more flexible rule representation than ID3, so the rules produced should be simpler than those produced by ID3, and nearly as good as those produced by AQ11. An example using an implementation of the SPLIT algorithm is given in Appendix C.

4.3. Direct Simplification

Since the covers produced by the generalization and specialization operations used in incremental hypothesis generation are not always optimal, redundant terms may be introduced into class covers. Although some simplification operations may be computationally expensive, the cost of most simplification operations is independent of the number of classes, and they may be applied selectively to reduce the impact of this expense on program performance. Several operations are discussed in this section for direct simplification of DVL1 expressions.

4.3.1. ABSORB

Sometimes in a disjunction of complexes (a DVL1 expression) one complex will include all of the points in another complex. In this situation, we want to remove the smaller complex. For example, given the DVL1 expression:

$$[X = 1][Y = 2][Z = 3] \vee [X = 2][Y = 1] \vee [X = 1][Y = 2]$$

the first complex may be deleted since it is included by the third complex, yielding the simpler expression:

$$[X = 2][Y = 1] \vee [X = 1][Y = 2].$$

The ABSORB algorithm consists of running two pointers through an expression and deleting the redundant complexes. It is quite simple and requires only an INCLUDES predicate to determine if one complex includes another:

```
ABSORB (expression)
  for ComplexA in expression
     for ComplexB in expression
        if (ComplexA $\neq$ ComplexB) and INCLUDES (ComplexA, ComplexB)
          then delete ComplexB from expression
     next ComplexB
  next ComplexA
```

Intracies of the data structure used may require an actual implementation to be more complicated, but this is the basic outline. In the worst case, when no simplification is possible, the computational cost of this algorithm is proportional to $A * N^2$, where A is the number of attributes used to represent events and N is the number of complexes in the expression. The ABSORB operation is especially useful for removing redundant complexes from partial stars in the TRUNCATE function used in the AQ algorithm.

4.3.2. INDEPENDENT

The INDEPENDENT operation removes complexes from a class cover in DVL1 form which do not cover any events in the class *independently*. The ABSORB operation is a special (and more efficient) case of the INDEPENDENT operation where one complex must be completely included by another to be considered redundant. If the disjunction of several complexes within an expression covers all of the events covered by some other complex, the non-independent complex should be deleted. For example, given the DVL1 expression containing complexes C1, C2 and C3:

$$C1 \vee C2 \vee C3$$

and events E1, E2, E3 and E4 such that:

C1 covers E1 and E2,
C2 covers E2 and E3, and
C3 covers E3 and E4,

C2 may be deleted because it does not cover any events independently, yielding the expression:

$$C1 \vee C3.$$

The algorithm proceeds by repeatedly evaluating the number of events covered independently by each complex, and deleting a non-independent complex. A set of complexes may be mutually non-independent, so the "worst" non-independent complex - as determined by specific quality criteria - should be deleted first. For this algorithm we assume "CoveredEvents" is a list of lists of events covered by each complex with the sublists in order corresponding to the order of complexes in "Cover".

The INDEPENDENT algorithm is outlined as follows:

```
INDEPENDENT (Cover, CoveredEvents)
  Repeat
    for Complex in Cover as CEvents in CoveredEvents collect
      (COUNTEVENTS (LISTDIFFERENCE CEvents
                        (APPENDX (REMOVE CEvents CoveredEvents))))
    end
    assign the list of counts to the variable IndepCount.
    NonIndependent := List of complexes in Cover for which IndepCount = 0.
    WorstComplex := the "worst" complex in NonIndependent.
    Remove WorstComplex from cover.
  Until (the number of NonIndependent complexes = 0).
```

Where,

COUNTEVENTS - returns the number of events in a list of events,

LISTDIFFERENCE - returns all members of its first argument which are not present in its second argument,

APPENDX - changes a list of lists of events to a list of events, and

REMOVE - returns its second argument with all occurences of its first argument removed.

The computational cost of this algorithm is roughly proportional to $(C + 1) * E^2$ where C is the number of non-independent complexes and E is the total number of events covered by the cover. The E^2 factor is due to the LISTDIFFERENCE operation, and the (C + 1) factor corresponds to the number of iterations of the Repeat loop.

4.3.3. MERGE

The MERGE operation changes external disjunction in DVL1 expressions to internal disjunction where possible. For example,

$$[X = 1][Y = 2][Z = 3] \vee [X = 1][Y = 2][Z = 1]$$

may be simplified to

$$[X = 1][Y = 2][Z = 1,3].$$

Two complexes may be merged if they differ by no more than one selector. The complexes are merged by taking their REFUNION. The REFUNION operation simply forms the logical union (OR) of values on the right-hand-side of corresponding selectors.

The basic outline of the MERGE algorithm is as follows:

```
MERGE (Expression)
  for ComplexA in Expression
    for ComplexB in Expression
      if #-of-Differenct-Selectors (ComplexA, ComplexB) $\leq$ 1
        then replace ComplexA by REFUNION (ComplexA, ComplexB),
          delete ComplexB from Expression.
    next ComplexB
  next ComplexA
```

In the worst case, the computational cost of the MERGE algorithm is roughly proportional to $A * N^2$ where A is the number of attributes used to describe objects and N is the number of complexes in Expression.

4.3.4. Conclusions

Some techniques for direct simplification of DVL1 expressions have been discussed and algorithms given. The ABSORB operation is currently used in AQINTERLISP to improve efficiency of the star generation process. INDEPENDENT and MERGE will be used in the near future in implementations with incremental hypothesis generation capabilities. Some simple metrics must yet be devised for determining when to apply these simplification operations.

5. Beyond Incremental Hypothesis Generation

So far the discussion has centered around incremental hypothesis generation. This is just a first step in the area of incremental learning of discriminant descriptions. What we have discussed is how to modify discriminant class descriptions (covers) to maintain consistency with new training instances (events). A system should also be capable of dealing with new classes, new or deleted attributes, and modifications to attribute domains. In addition, a system should be capable of dealing with hierarchies of classes, non-monotonic sets of training instances, and noisy data.

5.1. Adding New Classes, Changing the Event Space

Adding new classes to a system is relatively simple to deal with. The same operators which are used to modify existing class descriptions when new training events are presented may be used when new classes are presented. Existing class covers which cover events in the new class(es) must be specialized and a cover must be generated for the new class.

In a real world problem, it is not always known what parameters should be measured to acquire information in a specific problem area. The ability to obtain certain measurements may be gained or lost over time. Thus, a system should be capable of handling new or deleted attributes. This is in fact a rather sticky problem. If a new attribute is introduced it will be of little value for forming discriminations unless values for that attribute are entered for previously presented examples. If an attribute is deleted, for whatever reason, it will be necessary to re-evaluate all rules which use that attribute for making a discrimination. Preferably, an attribute should be dropped only after it has been determined to be redundant. The only reason to drop redundant attributes is to save system and user time. Attribute domain changes should be handled in a similar fashion, with extensions allowed at any time, but reductions allowed only when an attribute value is determined never to occur.

Of greater importance are derived attributes (and meta-attributes) which are formed by a process known as "constructive induction". Sometimes, several attributes may have meaningful interactions. For example, length, width, and breadth of a rectangular object may be combined via a simple mathematical relation to find volume. Volume may then turn out to be a promising attribute for forming discriminant descriptions. One of the most successful techniques to date has been to explicitly code such information about relations between attributes in "background rules". Syntactic pattern matching is used to determine if a background rule may be applied to an event, and if it does, the attribute derived using the rule is added to the event. Generally, the process of adding derived descriptors is performed before any inductive generalization processes. An example of a system which uses background rules in this manner is the INDUCE2 system which is described in [Hoff, Michalski, and Stepp, '83]. Lenat uses heuristics in a similar fashion to add attributes to concepts by creating new slots in a frame based system [Lenat, '83].

5.2. Hierarchical Learning

As previously mentioned, discriminating between members of different classes may be viewed as a problem of partitioning a class into subclasses. Since most real world information may be conveniently organized into hierarchies we would like our systems to be able to learn such hierarchies. Common examples are plant and animal taxonomies and business organization. The VL1 representation language may be used for hierarchical learning, but this would be restrictive and awkward for complex problems. A system which includes frame-like objects, such as the anotated predicate calculus, would be more suitable.

Many current inductive learning systems provide limited hierarchical learning facilities in the form of STRUCTURED variables. With this type of facility, climbing generalization may be performed on single attributes. But, the user must supply the structure in a domain declaration, so the structure itself is not actually learned. We would like the structure to be learned from information present in training examples.

A system should be able to build class hierarchies both in the upwards (more general) and downwards (more specific) dirctions. For example, say that the system has learned concept descriptions for birds, mammals, reptiles, fish, trees, grasses, algea, and so on. We should then be able to teach the concept "animals" in terms of these higher level concepts. The system should create rules which are stated in terms of higher level concepts whenever possible. As an example of building downward, say that the system has learned to distinguish birds form other types of animals. We should then be able to teach the system to distinguish between kinds of birds. Basic techniques for learning discriminant descriptions can be used for building hierarchies in both directions.

5.3. Non-monotonicity and Noise

Events previously presented to a learning system may later be found to be wrong (or unsubstantiated). A learning system should have the capability of deleting old training events which turn out to be wrong. Class covers may then be adjusted using the specialize/generalize mechanism used for making other incremental changes. So, this type of non-monotonicity is easily handled.

Noisy data is bound to occur in nearly every learning situation. Generally, noise events will be covered by "low weight" complexes in a class cover, i.e. complexes which cover few events. Or, noise events may appear as contradictory information. It may be possible to use similarity measures, such as sparseness and N-promise, to detect noise events and allow filtering them out. For example, given an event which is representative of more than one class in a problem where classes are assumed to be disjoint, a system could use similarity measures to determine to which class the event is most similar. It could then remove the event from the class to which it is least similar.

Given a complex in a class cover which covers relatively few events, a system could use similarity measures as above and possibly background knowledge (not necessarily Induce style background rules) to determine if the event(s) covered are irregular class members or noise. The solution may involve setting various thresholds, determining relationships between attributes, and having the program "know" about the salience of attributes used in the classification and the classification goals. For example, the program should be able to correctly classify a penguin as member of the class "bird" rather than as noise.

Some assumptions are necessary:

(1) The number of noise events is relatively few.

(2) There is high similarity between members of a class, and low similarity between members of different classes.

(3) The information available to the system can be made sufficient so that irregular class members can be distinguished from noise.

6. Learning Situation-Action Rules

As mentioned in the introduction, two applications of the methodology currently being discussed are statistical analysis and automatic generation of rules for expert systems from examples. The QUIN (QUery and INference) system is a combination of a number of other systems including GEM, AQ11, CLUSTER/2, ESEL, and PROMISE [Spackman, '83]. These systems are integrated in a statistical analysis tool based on relational data base techniques. The ADVISE group has been working on using rules generated by machine learning techniques in an expert medical diagnosis system, and the PLANT/ds system uses rules generated by machine from examples for plant pathology.

Work done with many expert systems and research done by Lenat on many aspects of heuristics has shown that heuristic knowledge is one of the most useful forms of knowledge. Heuristics may take the form of "if (situation) - then (action)" pairs. This is a very general form of decision rule which may be used in essentially any problem domain. A system capable of learning discriminant descriptions from examples could be used to learn such rules for a finite set of actions. When a system for learning situation-action rules is combined with a production system (a select-execute mechanism) and a feedback mechanism we have a very interesting "general purpose" system with both learning and task performance capabilities.

To be more specific, a *situation* is an object defined in terms of a set of attributes. These attributes may correspond to properties of physical or conceptual objects as with previous classification tasks involving structural descriptions. The attributes may also correspond to a current goal, variables in a production system "blackboard" memory, past actions, sensory information, etc. An *action* may be any action normally allowed in a production system (i.e. virtually anything) such as modifying global memory, sending a signal, constructing new productions, etc. The sets of attributes, attribute domains, and actions are user defined to fit the problem solved. Feedback may be achieved in the following way. Assume (for simplicity) that the system is single-stepping through a sequence of actions. Actions may change the current situation, so situations may occur which were not included in the training set. When an action is selected, the critic may respond and indicate that the choice was correct or incorrect, so the new situation may be treated as a positive or negative training example for the action selected. Of course, more complicated schemes may be developed for assigning credit and blame over long sequences of actions.

A simple example in the field of robotics was given in the introduction. Now consider the DENDRAL system for mass spectrogram analysis. The number of possible chemical formulas is so large that it would be virtually impossible to learn a discriminant description for each chemical formula from examples, as is currently done in PLANT/ds for soybean diseases. The task must be broken down into a number of smaller steps.

"The DENDRAL system works out structure from chemical formulas and mass spectrograms using the following steps:

- The mass spectrogram is used to create lists of required and forbidden substructures.
- The chemical formula is fed to a structure generator capable of generating all possible structures. The structure generator limits its output to things consistent with the lists of required and forbidden substructures.
- The mass spectrogram is predicted for each structure generated.
- The generated mass spectrograms are all compared with the actual experimental spectrogram. The correct structure is the one whose generated spectrogram gives the best match." [Winston, '79]

For the first step, "For any given category of organic chemicals, the keytones or estrogens for example, there are about six to ten ... rules." [Winston, '79] The rules add items to the lists of required and forbidden substructures. About 100 productions are used in the third step to predict mass spectrograms from chemical structures. Other expert knowledge is used in the other steps. All of the expert knowledge in this system, encoded as productions (situation-action rules) could be learned incrementally from examples and feedback from experts. Most likely, the learning task would be broken into segments corresponding to the steps given above.

Current methodologies are adequate for initial experimentation in this area. To be robust, a system used for learning situation-action rules should have strong constructive induction, incremental learning and hierarchical knowledge learning capablilities.

7. Conclusions

Several topics in the area of incremental learning of discriminant descriptions have been discussed. Previous approaches have been discussed, and some of their strengths and weaknesses identified. The problem of lopsidedness has been discussed and possible solutions identified. Several new, and not so new, operators useful for incremental learning of discriminant descriptions have been discussed and demonstrated. Important issues beyond incremental hypothesis generation have been discussed, and the application of current methodologies to the learning of situation-action rules has been proposed.

References

(1) -, Interlisp Reference Manual, Warren Teitelman, ed., Xerox Palo Alto Research Center, California, 1978.

(2) -, Interlisp-D User's Guide, Xerox Electro-Optical Systems, Pasadena, California, February 1982.

(3) Baim, Paul W., "The PROMISE Method for Selecting Most Relevant Attributes For Inductive Learning Systems", File # UIUCDCS-F-82-898, ISG 82-1, Department of Computer Science, University of Illinois, Urbana, September 1982.

(4) Baim, Paul W., "Automated Acquisition of Decision Rules: The Problems of Attribute Construction and Selection", Masters Thesis, Department of Computer Science, University of Illinois, Urbana, 1983.

(5) Barsalou, L. W., "Ad hoc categories", Memory & Cognition, Vol. 11, No. 3, pp 211-227. The Psychonomic Society, Inc., Robert A. Bjork, Editor, May 1983.

(6) Barsalou, L. W., "Context-Independent and Context-Dependent Information in Concepts", Memory & Cognition, Vol 10, No. 1, pp 82-93. The Psychonomic Society, Inc., Robert A. Bjork, Editor, January 1982.

(7) Burstein, Mark H., "Concept Formation by Incremental Analogical Reasoning and Debugging", *Proceedings of the International Machine Learning Workshop,* Ryszard S. Michalski, Editor, Department of Computer Science, University of Illinois, Urbana. June 22-24, 1983, pp 19-25.

(8) Carbonell, Jaime G., "Derivational Analogy in Problem Solving and Knowledge Acquisition", *Proceedings of the International Machine Learning Workshop,* Ryszard S. Michalski, Editor, Department of Computer Science, University of Illinois, Urbana. June 22-24, 1983, pp 12-18.

(9) DeJong, Gerald, "An Approach to Learning From Observation", *Proceedings of the International Machine Learning Workshop,* Ryszard S. Michalski, Editor, Department of Computer Science, University of Illinois, Urbana. June 22-24, 1983, pp. 171-176.

(10) Dietterich, Thomas G. and Ryszard S. Michalski, "A Comparative Review of Selected Methods for Learning from Examples", Chapter 3 in the book *Machine Leaning, An Artificial Intelligence Approach,* R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Editors, Tioga Publishing Company, Palo Alto, California, 1983, pp. 41-82.

(11) Hoff, William A., Ryszard S. Michalski, and Robert E. Stepp, "INDUCE2: A Program for Learning Structural Descriptions from Examples", File # UIUCDCS-F-83-904, ISG 83-4, Department of Computer Science, University of Illinois, Urbana, January, 1983.

(12) Lenat, Douglas B., "The Role of Heuristics in Learning by Discovery: Three Case Studies", Chapter 9 in the book *Machine Leaning, An Artificial Intelligence Approach,* R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Editors, Tioga Publishing Company, Palo Alto, California, 1983, pp. 243-306.

(13) Michalski, R. S., "A Theory and Methodology of Inductive Learning", Artificial Intelligence 20, pp. 111-161, North-Holland Publishing Co., Amsterdam, 1983.

(14) Michalski, R. S., "Variable-Valued Logic: System VL1", IEEE Catalog Number 74CH0845-8C, ONR Identifying Number NR 048-616, from the Prodeedings of the 1974 International Symposium on Multiple-Valued Logic, Morgantown, West Virginia, May 29-31, 1974.

(15) Michalski, R. S., J. H. Davis, V. S. Bisht, and J. B. Sinclair, "PLANT/ds An Expert Consulting System for the Diagnosis of Soybean Diseases", Accepted for publication in *Plant Diseases* and *Proceedings of the First European Conference on Artificial Intelligence,* Orsay, France, July 12-14, 1982, pp. 133-138.

(16) Michalski, R. S., and J. B. Larson, Revised by K. Chen, "Incremental Generation of VL1 Hypothesis: the underlying methodology and the description of program AQ11", File No. UIUCDCS-F-83-905, ISG 83-5, Department of Computer Science, University of Illinois, Urbana, January, 1983.

(17) Michalski, Ryszard S., and Robert E. Stepp, "How to Structure Structured Objects", *Proceedings of the International Machine Learning Workshop,* Ryszard S. Michalski, Editor, Department of Computer Science, University of Illinois, Urbana. June 22-24, 1983, pp. 156-160.

(18) Michalski, Ryszard S., and Robert E. Stepp, "Learning From Observation: Conceptual Clustering", Chapter 11 in the book *Machine Leaning, An Artificial Intelligence Approach,* R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Editors, Tioga Publishing Company, Palo Alto, California, 1983, pp. 331-364.

(19) O'Rorke, Paul O. "A Comparative Study of Two Inductive Learning Systems", Internal Report, Department of Computer Science, University of Illinois, February 1982.

(20) Quinlan, Ross J., "Learning Efficient Classification Procedures and their Application to Chess End Games", Chapter 15, pp 463-482 in the book "Machine Leaning, An Artificial Intelligence Approach", R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Editors, Tioga Publishing Company, Palo Alto, California, 1983.

(21) Rychener, Michael D., "The Instructible Production System: A Retrospective Analysis", Chapter 14, pp 429-460 in the book "Machine Leaning, An Artificial Intelligence Approach", R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Editors, Tioga Publishing Company, Palo Alto, California, 1983.

(22) Spackman, Kent A., "QUIN: Integration of Inferential Operations in a Relational Database", Master's Thesis, Department of Computer Science, University of Illinois, Urbana, 1983.

(23) Winston, Patrick H., *Artificial Intelligence,* Addison-Wesley Publishing Co., Inc., April 1979.

(24) Winston, Patrick H., "Learning by Augmenting Rules and Accumulating Censors", *Proceedings of the International Machine Learning Workshop,* Ryszard S. Michalski, Editor, Department of Computer Science, University of Illinois, Urbana. June 22-24, 1983, pp. 2-11.

Appendix A

This appendix provides an introduction to some of the terminology used in this paper. Other terminology is explained as it is used. A full, formal description of the VL1 variable-valued logic system may be found in [Michalski, '74].

Variables are one of the basic representational units in VL1. Associated with each variable is a *domain* which defines which *values* the variable may assume. Four basic types of domains are now in common use: NOMINAL, LINEAR, STRUCTURED, and CYCLIC. A NOMINAL domain includes only discrete values, such as "sweet", "sour", and "salty" for the variable "flavor". A LINEAR domain includes an interval of linearly ordered values, such an interval in the integers or real numbers. A STRUCTURED domain includes discrete values in a tree structure. For example, "triangle", "square", and "hexagon" may be siblings of "polygon" in a STRUCTURED domain for variable "shape". A CYCLIC domain includes an interval of linearly ordered values where the first element may be considered to be a successor of the last, such as the days of the week.

Relations associate variables with values. Typical relations are $=$, $\neq$, $<$, and $>$. A *selector* has the form:

[variable relation value(s)].

A selector is *satisfied* if the indicated condition is met. Selectors are commonly used for representing attribute-value pairs or *feature vectors*. Some examples of selectors are:

Selector	English Phrase
[color $\neq$ red]	color is not red
[temperature = 50 .. 100]	temperature is 50 to 100
[size = small, large]	size is small or large

An event space is defined by a set of variables and their associated domains. Each point in the event space corresponds to an allowed combination of variables and values. A point in an event space may be described by a conjunction of selectors and is called an *event*. A conjunction of selectors which covers more than one point in an event space is called a *formula*. Events and formulas are both *complexes*. A complex is any conjunction of selectors. A union of complexes is called a disjunctive normal form (DNF) expression or a DVL1 expression.

The *product* of two complexes is essentially those points in the event space which are contained in the intersection of the two complexes. The *difference* of two complexes C1 - C2 is the intersection of C1 with all points **not** contained in C2. The union of two complexes is as one would expect.

Appendix B

In this appendix, the top level of the basic AQ algorithm is given. The parameter EList is a list of complexes to be covered. The parameter FList is a list of complexes which must not be covered. AQ returns the cover of EList against FList.

```
AQ (EList, FList)
  UnCoveredEList := EList
  Repeat
    Seed := randomly chosen event from UnCoveredEList
    Star := STAR (Seed, FList)
    BestComp := BESTCOMP (Star)
    Cover := Cover ∪ BestComp
    UnCoveredEList := KNOCKOUT (UnCoveredEList, BestComp)
  Until (UnCoveredEList = NIL)
  Return (Cover).
```

Where,

STAR - returns the reduced star of Seed against FList. This procedure is described in more detail below.

BESTCOMP - chooses the best complex from a star based on specific quality criteria.

KNOCKOUT - removes complexes from its first argument which are covered by its second argument.

The STAR function returns the reduced star of a seed complex E against a list of negative examples FList. A star is "a set of all possible alternative nonredundant descriptions of *e* that do not violate constraints *F* " [Michalski, '83]. A more restricted type of star is actually produced which consists of a set of maximally general complexes which cover event E and do not cover any of the negative events F. A reduced star consist of no more than some fixed number of the most preferable descriptions, as determined by size limit MaxStar and specific preference criteria.

```
STAR (E FList)
  Product := (NIL)
  for F in FList
    Product := MULTIPLY (Product, EXTENDAGAINST (E, F))
    Product := TRUNCATE (Product)
  end
  Return (Product)
```

Where,

EXTENDAGAINST - returns a list of selectors which are extensions of selectors in complex E against corresponding selectors in complex F.

MULTIPLY - forms the product of each complex in its first argument with each selector in its second argument.

TRUNCATE - reduces the number of complexes in Product to below some fixed number (MaxStar) by selecting only the most preferable complexes.

Appendix C

This appendix contains some samples rules produced by current implemntations of AQ and SPLIT. SPLIT uses a specialized version of N-promise for attribute selection. The first example is a comparison of rules produced using single-class generation versus multi-class generation, and corresponding computation times. The example given illustrates the points made in the text concerning lopsidedeness and speed. On the average, multi-class generation gives better performance than single-class generation. However, it should be pointed out that multi-class generation may at times be slower and will produce more complicated rules than single class generation due to the structure of the data set being processed. The following data set, consisting of some peculiar animals, was used:

Domain Declarations

Variable	Domain Type	Value Set
Blood-Temp	NOMINAL	Hot, Cold
Body-Covering	NOMINAL	Fur, Feathers, Scales, Skin
Habitat	NOMINAL	Air, Water, Land
Milk	NOMINAL	Yes, No
Eggs	NOMINAL	Yes, No
Weight	LINEAR	1 .. 300

Events

Class	Blood-Temp	Body-Covering	Eggs	Habitat	Milk	Weight
Birds	Hot	Feathers	Yes	Air	No	1
	Hot	Feathers	Yes	Air	No	10
	Hot	Feathers	Yes	Land	No	50
	Hot	Feathers	Yes	Water	No	20
	Hot	Skin	Yes	Air	No	200
Mammals	Hot	Fur	No	Land	Yes	175
	Hot	Fur	No	Land	Yes	50
	Hot	Fur	No	Land	Yes	10
	Hot	Fur	Yes	Water	Yes	20
	Hot	Skin	No	Water	No	300
Reptiles	Cold	Scales	Yes	Land	No	3
	Cold	Scales	Yes	Water	No	75
	Cold	Scales	Yes	Land	No	20
	Cold	Skin	Yes	Water	No	1

AQINTERLISP was run the CRL VaxB at the University of Illinois Department of Computer Science in both single-class and multi-class generation modes. Note that multi-class generation is slightly faster and the covers for Mammals and Reptiles are less lopsided. In the single-class case, the ratio of the number of points in the event space covered by the Mammals class cover to the number of points in the event space covered by the Reptiles class cover is 4:1. In the multi-class case it is 4:3.

Using **single-class** generation the following covers were found:

Cover of Class: Birds
[Blood-Temp = Hot][Body-Covering = Feathers, Skin][Eggs = Yes]

Cover of Class: Mammals
[Body-Covering = Fur] v [Eggs = No]

Cover of Class: Reptiles
[Blood-Temp = Cold][Body-Covering ≠ Fur][Eggs = Yes]

Computation Time: 2.016 CPU sec.

Using **multi-class** generation the following covers were found:

Cover of Class: Birds
[Blood-Temp = Hot][Body-Covering = Feathers, Skin][Eggs = Yes]

Cover of Class: Mammals
[Body-Covering = Fur] v [Blood-Temp = Hot][Eggs = No]

Cover of Class: Reptiles
[Blood-Temp = Cold][Body-Covering ≠ Fur]

Computation Time: 1.792 CPU sec.

The current version of the SPLIT algorithm will of course be subject to further refinements. Preliminary results using the current implementation of SPLIT (which is not highly optomized) indicate that SPLIT is more costly in terms of CPU time than AQINTERLISP for small problems, but that the cost does not grow as rapidly as problem size increases. This agrees with cost estimates given in the text. Further emperical testing is needed to determine at what point (if at all) SPLIT becomes less costly then AQINTERLISP. Other revisions to the SPLIT algorithm are needed to allow it to produce less lopsided disjoint covers with the same representational flexibility as is provided by AQ.

The covers formed by the SPLIT program on the same data set are quite lopsided:

Cover of Class: Birds
[Body-Covering = Feathers] v [Body-Covering ≠ Feathers][Habitat = Air]

Cover of Class: Mammals
[Blood-Temp = Hot][Body-Covering = Fur, Skin][Habitat ≠ Air]

Cover of Class: Reptiles
[Blood-Temp = Cold][Body-Covering = Scales, Skin][Eggs = Yes][Habitat ≠ Air][Milk = No]

Computation Time: 2.64 CPU sec.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-F-85-935	2.	3. Recipient's Accession No.

4. Title and Subtitle Topics in Incremental Learning of Discriminant Descriptions	5. Report Date February 1985
	6.
7. Author(s) Jeff Becker	8. Performing Organization Rept. No.
9. Performing Organization Name and Address Department of Computer Science University of Illinois Urbana, IL 61801	10. Project/Task/Work Unit No.
	11. Contract/Grant No. NSF DCR 84-06801
12. Sponsoring Organization Name and Address National Science Foundation Washington, DC	13. Type of Report & Period Covered
	14.

15. Supplementary Notes

16. Abstracts

Discriminant descriptions may be learned from classified sets of examples of objects. These descriptions may be used as decision rules. In many situations it is desirable to be able to easily modify existing rules to make them consistent with newly observed examples. This paper covers a number of topics in the area of learning discriminant descriptions from examples. Topics covered include lopsidedness, the N-promise algorithm, the SPLIT algorithm, direct simplification, constructive induction, learning hierarchies of class descriptions, and learning situation-action rules.

17. Key Words and Document Analysis. 17a. Descriptors

Machine Learning
Discriminant Descriptions
Expert Systems
Inductive Inference
Knowledge Acquisition

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 37
	20. Security Class (This Page UNCLASSIFIED	22. Price