A Survey of Induction Algorithms for Machine Learning

Bruce W. Porter

Technical Report 207

June 1983

Department of Information and Computer Science
University of California, Irvine

This research was supported by
contract N00123-81-C-1165 from the
Naval Ocean Systems Center.

ABSTRACT

Central to all systems for machine learning from examples is an induction algorithm. The purpose of the algorithm is to generalize from a finite set of training examples a description consistent with the examples seen, and, hopefully, with the potentially infinite set of examples not seen. This paper surveys four machine learning induction algorithms. The knowledge representation schemes and a PDL description of algorithm control are emphasized. System characteristics that are peculiar to a domain of application are de-emphasized. Finally, a comparative summary of the learning algorithms is presented.

1. Introduction

This paper examines the induction algorithms developed by the following research projects in the area of machine learning:

- Winston's ARCH program [23] which learns concepts to describe simple structures.
- Quinlan's ID3 program [19] which has been applied to the domain of chess endgame win/loss classification.
- Michalski, et.al. INDUCE1.2 program [14] which has been applied to several domains including bean disease classification and structural descriptions.
- Mitchell, et.al. LEX program [17] which learns heuristics for problem solving in the domain of integration.

This paper focuses on the central component of these learning systems: the induction algorithm. In section 2, we examine each algorithm from a domain independent point of view. In section 3, we compare the algorithms with emphasis on relative strengths and weaknesses important for performance.

2. Descriptions of Induction Algorithms

In this section we describe the induction algorithms of four machine learning research projects. By abstracting to a domain independent algorithm, we hope to illuminate commonalities and differences.

2.1. Assumptions and Format

Each algorithm is presented with its knowledge representation scheme and a PDL description of the processing. The important knowledge for the induction algorithm is represented in the instance and generalization languages for expressing examples and concepts, respectively. Since the induction algorithms are being studied apart from the overall architecture of each learning system and without concern for the domain used to test it, we make the following assumptions:

- Training instances are presented to the induction algorithm and are pre-classified by concept.
- Description languages are adequate for representing training instances and concept definitions.[1]
- The training session is noise-free. In section 3 we examine whether this constraint can be relaxed for each algorithm.

2.2. Winston's ARCH Algorithm

Similar to other induction algorithms, ARCH allows both unary and binary relations in descriptions. The unary relations (or features) describe facts about a single object (e.g. its color or shape). Binary predicates express relationships between two objects (e.g. relative positions).

Winston uses a semantic net representation for both the instance language and the generalization language. Arcs are annotated with names of binary relations between nodes which represent objects. In addition, nodes are created with names of unary relations which are attached via HAS-PROPERTY-OF arcs to the objects they describe.

The single representation technique [16] is used, which means that the generalization language contains the instance language. For arcs, the generalization language also includes annotation which expresses the relative importance of a relationship to the correct concept definition. For example, in the instance language, an arc name might be a relationship between two objects in the instance. In the generalization language, this relationship may be emphatic (MUST-BE, MUST-NOT) or optional (MAYBE). For nodes, the additions consist of a renaming of object names to form a "generic" object.

[1]Some initial work has been done on introducing new descriptors to enable generalization. See [10, 14, 21].

Given two training instances, corresponding objects (nodes) from each semantic net form a common name in the generalization.

A third representation is used by Winston to express the differences between a training instance and the current concept description. This is also a semantic net. Objects considered irrelevant to the concept are simply deleted in this representation. Moreover, meta-relations (i.e. relations between relations) are represented. For example, if two nets, SN_1 and SN_2, are compared and relation R holds in SN_1 and relation ~R holds in SN_2, then the relation OPPOSITE can be used to describe the difference.

Given the description of the differences between a training instance and the current concept definition, the algorithm specializes (for negative instances) or generalizes (for positive instances) the concept description. Generalization/specialization operators that are used are:

1. climbing/descending concept tree - meta-relations are related by strength using a concept tree. Also objects are generalized (e.g. a CUBE is a type of BLOCK).

2. dropping conditions - objects considered irrelevant to the concept are dropped by removing them from the semantic net. Note that relationships found irrelevant are simply weakened by (1).

Winston´s induction algorithm is described in figures 2-1 - 2-3.

```
Concept <-- first positive TI
REPEAT
  Input a TI
  Diff <-- skeletal_match(Concept,TI)
  Concept <-- incorporate_diff(Concept,Diff)
  DISPLAY Concept
UNTIL teacher satisfied
```

Figure 2-1: Winston´s Main Loop

```
FUNCTION skeletal_match(Concept,TI)
  skeleton <-- match nodes of TI and Concept semantic nets
    to find "best" correspondences[2].
  skeletal_match <-- annotate skeleton with notes describing
    the match.  Most common is INTERSECTION which means both
    matched nodes in the two graphs point to the same concept
    (node) with the same relationship (arc).  Another note is
    NEGATIVE-SATELLITE used to denote opposite relationships.
    EXIT notes annotate nodes outside of the match.  These
    nodes are considered irrelevant to the concept.
```

Figure 2-2: Skeletal Match Function

```
FUNCTION incorporate_diff(Concept,Diff)
  incorporate_diff <-- heuristically selected "best"
    generalization of concept with Diff.[3]
    Non-determinism arises in the matching process and
    the specialization of the concept.  Generalizing with
    positive instances is not a problem because they loosen
    constraints by minimal generalization of relations using a
    concept tree.  Thus any number of differences can exist
    between the current concept and a positive instance (as
    long as a skeletal match can be found).
    Negative instances are a problem because:
       1) there are multiple features of a training instance
          which might account for the "miss"
       2) there are multiple ways to specialize a relation
          using the concept tree.
    Only "near-misses" are permitted for negative instances.
    A near miss is defined [24] as differing from the
    current concept in only one feature. This constraint
    mitigates these problems.
```

Figure 2-3: Incorporate Difference Function

[2] Sub-graph isomorphism (matching) is NP-complete. However the search is constrained by arc annotation. Winston ignores the problem of multiple matchings by expecting the teacher to present examples with little difference. For one approach to handling "interference matching" see [5].

[3] This is a point of non-determinism. The algorithm backtracks if the generalization returned is found to violate subsequent training instances.

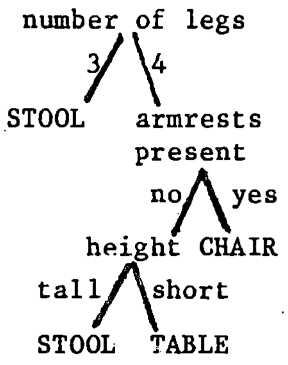
2.3. Quinlan's ID3 algorithm

The goal of this induction algorithm is to produce a concept definition which can efficiently classify instances. To this end, Quinlan (see also [7]) builds a decision tree from a collection of pre-classified training instances. This tree is analogous to a compiled collection of rules.

The instance language for ID3 is a conjunct of features. If n features are used to classify instances then the feature vector is of the form:

$<f_1,f_2,\ldots,f_n>$ s.t. for all $1<=i<=n$
f_i is an element of $\{v_{i1},v_{i2},\ldots,v_{im}\}$, the set of m possible values for feature i.

The generalization language is a decision tree which is equivalent to a collection of rules each with conjunction and disjunction of feature values. The nodes represent features chosen from the set $\{f_1,f_2,\ldots,f_n\}$ and the arcs from a node labeled f_i are chosen from the set of values for f_i, $\{v_{i1},v_{i2},\ldots,v_{im}\}$. Leaves of the tree are labelled with names from the set of classes, $\{c_1,c_2,\ldots,c_k\}$. Figure 2-4 is an example of a decision tree which ID3 can form.



```
classes={chair,stool,table}
features={number of legs,
          armrests present,
          height}
values: number of legs={3,4}
        armrests present={yes,no}
        height={tall,short}
```

Figure 2-4: Example ID3 decision tree

An instance is classified by the decision tree by traversing the tree from the root. At each internal node labeled f_i, evaluate the instance with respect to f_i and traverse the arc labelled with the result of the evaluation. When a leaf is reached then the instance is classified by the leaf's value.

The induction algorithm is surprisingly straightforward.[4] The PDL is shown in figures 2-5 - 2-6.

```
Instances <-- a set of training instances
Features <-- feature vector <f_1,f_2,...,f_n> input from teacher
Domains <-- set of domains for Features.  {d_1,d_2,...,d_n}
            where each d_i is a set of possible values
            {v_1i,v_2i,...,v_mi} for f_i.
Classes <-- set of classes {c_1,c_2,...c_k}.  Note this
            set is simply {+,-} for single concept learning.
Rule <-- form_rule(Instances,Features,Domains,Classes)
DISPLAY Rule
```

Figure 2-5: ID3 Main Program

```
FUNCTION formrule(Instances,Features,Domains,Classes)
  For some class in Classes, IF all members of Instances
     fall into class then RETURN class
  ELSE
    f <-- select_feature(Features, Instances)
    d <-- domain from set Domains corresponding to f
    RETURN a tree of form:

             f          <-- root labeled f
       d_1/ |d_2 \ d_m  <-- arcs labeled with
         /  | . .\          values from d
                        <-- each child is a subtree
                        created by recursive call:
 formrule({i|i in Instances, eval(i,f)=d_j},Features,Domains)
        for 1<=j<=m
```

Figure 2-6: Recursive function FORMRULE

The select_feature function selects a feature from the set of features which will be used to sub-divide the current set of instances. Several of the more interesting criteria which might be used in the select_feature function

[4]The author implemented it using about 20 lines of Prolog (not including the beam search; see below).

are:

- random selection -- while guaranteed to result in a correct decision tree (i.e. complete and consistent with instances given),[5] the rule is sub-optimal. Features may be chosen which do not divide the set of instances into useful subsets. For example, all instances may share one value of the feature, resulting in a node with one descendant. Or, more subtle, the feature may subdivide the instances, but not be criterial to the rule. In this case the subsets have the same mixture of positive and negative instances as the set before subdividing.

- information theoretic selection -- as explained by Quinlan [19] this method selects the feature (for sub-dividing each node) which results in a tree with minimum expected classification time. The feature is selected which is most criterial to the concept being formalized. Criteriality is measured by the ability of a feature to classify instances.

- minimal cost selection -- used in Hunt's original CLS system [7], this feature selection method balances the cost of evaluating an instance for a feature with the cost of misclassifying the instance. Input to the function are two sets of costs: $\{P_i\}$ of measuring the i^{th} feature of some instance and $\{Q_{jk}\}$ of misclassifying it as belonging to class j when it is really a member of class k. The goal is to minimize the combined costs. This method is useful when the cost of evaluating instances is not uniform for all features (e.g. medical diagnosis [18]).

When the size of the set of Instances becomes large the goal of efficient rule generation is lost. The problem occurs in two steps of the formrule function: the first step of determining whether all members of Instances fall into the same class and the last step which involves evaluating each instance with respect to a feature. Due to this practical concern ID3 has been implemented using a beam search. Basically, a subset of the set of instances is selected and a rule is formed which is complete and consistent with respect to the subset. If the rule is not contradicted by any instances in the set of instances, then the beam search terminates. Otherwise, a new selection of

[5]As defined in [13], a concept is complete if it covers all past positive instances; A concept is consistent if it does not cover any past negative instances.

instances is made and the search continues. Quinlan [19] and Michalski [14] have experimented with different criteria for selecting the subset of instances to use at each iteration of the search.

2.4. Michalski et.al.´s INDUCE algorithm

The INDUCE algorithm, as described in [14], forms a generalization rule using a multitude of generalization techniques and a highly expressive description language. Both the instance language and the generalization language allow descriptions of an instance or a concept using Annotated Predicate Calculus (APC). Descriptions are of the form:

<quantifier form><conjunction of relational statements>

where <quantifier form> represents zero or more quantifiers and <relational statements> are predicates expressed in APC. Michalski defines APC with a set of syntactic additions to first-order predicate calculus (FOPC) and semantic preserving two-way transformations. This allows for more natural encodings of knowledge. Examples of APC usage and mappings to FOPC are given in figure 2-7.

APC form	Example	APC <--> FOPC
internal conjunction	went(mary&mother,movie)	p(T1&T2) <--> p(T1)&p(T2)
internal disjunction	inside(key,(desk1vdesk2)	p(T1vT2) <--> p(T1)vp(T2)
relational stmts.	length(rect)>width(rect)	Term1 rel Term2 <--> rel(Term1,Term2)
negations of relational stmts.	~(size(ball)=large)	~(relational stmt) <--> <relational stmt with opposite relation>
exception operator (-\)	went(mary,movie) -\ went(mother,movie)	S1 -\ S2 <--> (~S2=>S1)&(S2=>~S1)

Figure 2-7: Examples of Description Language

The instance language and the generalization languages differ only in that internal disjunction is prohibited in the former and allowed in the latter. This a natural restriction since a known instance can be described without this non-determinism. Used in the generalization language, internal disjunction allows more efficient and "natural" representation than the traditional external disjunction used in Vere's Thoth system [22], for example.

The generalization rules employed by INDUCE are also very powerful. To some extent this power results from "big step" generalizations. More conservative generalization rules, such as climbing generalization tree, turning constants to variables and dropping a condition, proceed in small steps and are not as likely to drastically deviate from the "correct" learning path. Michalski introduces (where, following Michalski's notation, EXP stands for an arbitrary expression, ::> stands for "is in class", and |< stands for "can be generalized to"):

1. The closing the interval rule:

$$\left.\begin{array}{l}\text{EXP} \ \& \ [L=a] \ \ ::> K \\ \text{EXP} \ \& \ [L=b] \ \ ::> K\end{array}\right| < \ \text{EXP} \ \& \ [L=a..b] \ \ ::> K$$

   This rule states that two rules can be generalized if they differ only in the value of a term. The values are assumed to be extremes of a range for which the rule applies.

2. The extension against rule:

$$\left.\begin{array}{l}\text{EXP}_1 \ \& \ [L=R1] \ \ ::> K \\ \text{EXP}_2 \ \& \ [L=R2] \ \ ::> \sim K\end{array}\right| < \ [L \neq R2] \ \ ::> K$$

   The rule states that given two rules, one positive and one negative for concept K, form a generalization which ignores all but one term from each rule. The rule then infers that any instance for which descriptor L does not take value R2 is positive for class K.

3. Constructive generalization rule:

$$\left.\begin{array}{l}\text{EXP} \ \& \ F_1 \ \ ::> K \\ \qquad F_1 \ => F_2\end{array}\right| < \ \text{EXP} \ \& \ F_2 \ \ ::> K$$

   This rule generates an inductive assertion that uses descriptors (in this example F_2) not present in the original instance description. By applying background knowledge rules which draw conclusions from observed facts, new terms are introduced for use in the generalization language. This allows learned concepts to be used in forming new concepts. This incremental learning rule is also used by Sammut [20].

Using the description language APC and the powerful generalization rules discussed above, the INDUCE algorithm is summarized from [14] in figures 2-8 - 2-9.

```
pos <-- set of positive instances from teacher
neg <-- set of negative instances from teacher
m <-- upper bound on the number of candidate concept descriptions
REPEAT while COLLECTing all values of rule to form set rules
   posinst <-- a random selection from pos
   rule <-- formrule(posinst, pos, neg, m)
   reduce pos to include only those instances not covered by rule
UNTIL disjunction of elements of rules covers pos
apply collection of FOPC --> APC transformations on rules to
   get a simpler expression and DISPLAY
```

Figure 2-8: INDUCE main program

```
FUNCTION formrule(posinst, pos, neg, m)
   candidates <-- {f|f is one conjunct of posinst}
   order candidates, favoring those which cover the
     greatest portion of pos and reject the greatest
     portion of neg
   expand the set of candidates by applying the following
     inference rules to posinst:
      1) constructive generalization
      2) problem-specific generalizations defined for the
         domain (this can cover alot of "dirty-tricks" and
         is not well defined by Michalski)
      3) the definitions of previously-learned concepts to
         determine whether parts of posinst satisfy some
         already known concepts (again ill-defined).
   order candidates using criteria described above.
   delete all but the m most preferable descriptions from the
     set of candidates.
   solutions <-- {r|r is in candidates and r is complete and
     consistent with respect to pos and neg}
   consistent <-- {r|r is in candidates and r is consistent
     neg but incomplete with respect to pos}
   generalize elements of consistent by applying:
     1) extension against rule
     2) closing the interval rule
     3) climbing generalization tree rule
   add to the set of solutions those elements of consistent
     which are now complete
   formrule <-- m-best candidates from solutions as ordered
     by above preference criteria
```

Figure 2-9: INDUCE formrule function

Notice that the formrule function (figure 2-9) forms a set of maximally general rules which are complete and consistent with the positive and negative instances. This is due to the "seeds" chosen in the first step: maximal generalizations of a selected positive instance. These seeds are then specialized. Since the rules are maximally general, the INDUCE algorithm can form a rule to cover a (non-trivial) subset of the set of positive instances by considering only a single instance. One (maximally general) rule will then be found which covers the entire set.

2.5. Mitchell et.al.'s LEX algorithm

LEX, described in [16, 17], views concept learning as a search through a space of concept definitions. Both positive and negative training instances are used for navigating through the space. The space of candidates is structured by a partial ordering imposed on candidate concepts. This ordering is defined by a more-specific-than relation between concepts and results in a tree with the most specific concept at the root and least specific concepts at the leaves.[6] Positive instances force generalizations (i.e. searching deeper in a branch) and negative instances prune branches of the tree.

Traditional breadth first and depth first strategies for searching the space of candidate concept definitions require keeping past instances and verifying that current candidates do not violate previous instances. LEX avoids these time and space requirements by a search technique analogous to bi-directional search. LEX maintains two sets of candidate concept definitions. One set, S, contains candidates which are equal to or more

[6]If this ordering cannot be imposed on the generalization language for a domain then the LEX algorithm is not applicable.

specific than (as defined by the partial ordering) the correct concept. The other set, G, contains candidates which are equal to or more general than the correct concept. The correct concept lies between these boundaries. Mitchell calls this space between S and G (inclusive) the version space of the concept. The version space is implicitly defined and structured by concept hierarchy trees which relate terms in the description languages according to the partial ordering.

Provided to LEX is the set of concept trees necessary for representing instance descriptions and concept descriptions. The features used for an instance description must be contained in the leaves of the trees. The concept description must be expressible using the terms contained anywhere in the trees. An example concept tree for the concept FUNCTION is given in figure 2-10.

```
               function
                 /   \
           trans-     \
          cendental   polynomial
           /    \        /    \
      trig   explog  monomial  n-nomial
      / \     / \
   sin cos   exp ln
```

Figure 2-10: Concept Tree for FUNCTION

The LEX algorithm is described in figures 2-11 - 2-13.

```
Ctrees <-- set of concept trees from teacher
posinst <-- initial positive instance from teacher
S <-- {posinst}
G <-- {g|maximal_generalization(posinst,Ctrees,g)}
repeat
  TI <-- training instance from teacher
  if TI is a positive instance then
    retain in G only those elements g, s.t. match(TI,g,Ctrees)
    for all s in S, s.t. ~match(TI,s), replace s in S by
       generalize(s,TI,Ctrees)
    for all s in S, remove s from S if there exists g in G, s.t.
       match(s,g,Ctrees)
    for all distinct pairs of elements $s_1$ and $s_2$ in S, remove
       $s_1$ from S if match($s_2$,$s_1$,Ctrees)
  if TI is a negative instance then
    retain in S only those elements s, s.t. ~match(TI,s)
    for all g in G, s.t. match(TI,g,Ctrees), replace g in G by
       specialize(g,TI,Ctrees)
    for all g in G, remove g from G if there exists s in S, s.t.
       match(g,s,Ctrees)
    for all distinct pairs of elements $g_1$ and $g_2$ in G, remove
       $g_1$ from G if match($g_1$,$g_2$,Ctrees)
 until S=G
 Display S
```

Figure 2-11: LEX main program

```
FUNCTION generalize(concept, instance, Ctrees)
   RETURN the list of minimal generalizations of concept
   with the (ground) instance, w.r.t. Ctrees.
   Assuming a concept definition of n objects, the list of
   generalizations returned will be at most length n!.
   For each pairing of objects in concept with objects
   in instance, find the minimal generalization by:
     generalization <-- "null" term
     for all pairs of corresponding terms $t_i$ in instance and
     $t_c$ in concept
        ctree <-- member of Ctrees s.t. $t_i$ is a node of ctree
        append to generalization the closest common ancestor of
          $t_i$ and $t_c$ in ctree.

 FUNCTION specialize(concept, instance, Ctrees)
   RETURN the list of minimal specializations of concept
   with the (ground) instance, w.r.t. Ctrees.
   This list is formed by collecting the results of all successful
   paths through the non-deterministic algorithm:
     for all pairs of corresponding terms $t_i$ in instance
     and $t_c$ in concept
        ctree <-- member of Ctrees s.t. $t_i$ is a node of ctree
        replace $t_c$ with a descendant of $t_c$ from ctree
          in concept.
        if match(instance,concept,Ctrees) then
          concept<-- specialize(concept,instance,Ctrees)
        EXIT with concept
```

Figure 2-12: Generalize and Specialize functions

```
Boolean FUNCTION match(spec,gen,Ctrees)
  (comment: match is true iff spec is equal to, or more
   specific than, gen).
  RETURN true if
    for some pairing of objects in spec with objects in gen,
      for each pair of terms $t_s$ in spec and $t_g$ in gen,
      there exists ctree in Ctrees s.t.
        $t_s$=$T_g$ or
        $t_s$ is a descendant of $t_g$ in ctree
  otherwise RETURN false
```

Figure 2-13: Match test

The LEX induction algorithm terminates when the S and G boundaries meet. This convergence is guaranteed given sufficient (and different) training instances. If the boundaries pass then the training set must be noisy. The ability to estimate confidence in a partially-learned concept (as estimated by the "distance" between S and G) and to determine whether the training set was noisy is an important feature of LEX discussed in section 3.

3. Evaluations of Inductive Algorithms

In this section we evaluate the inductive algorithms described in section 2. Our main emphasis is on performance issues:

- adequacy of description languages -- what constraints do the instance and generalization languages place on concepts that can be learned?[7] Are the languages "natural" for representation?
- generalization rules -- do the generalization rules used by the induction algorithm restrict the class of concepts that can be learned? How rapidly does learning progress?
- complexity of induction algorithm -- what are the time and space requirements? Does the complexity restrict the class of concepts that can be learned?
- environmental requirements -- does the induction algorithm make assumptions about teacher behavior, peculiarities of the domain, or other external entity? What if the training set is noisy?

3.1. Winston´s ARCH Algorithm

Winston´s system represents instances and concepts with "simple" semantic nets. These are particularly useful for representing structure, which was Winston´s application domain. However, the description languages are weak. N-ary relations cannot be directly represented in semantic nets[8]. The

[7]As pointed out by [4], we are focusing on descriptions that can be learned, not just represented. For example, SPROUTER [5] uses a variant of FOPC as the representation language. Although disjunction can be represented in FOPC, only conjunctive concepts can be learned by this induction algorithm.

[8]Although not used by Winston, n-ary relations can be represented with n+1 binary relations. See [3].

instance language permits only conjunctive descriptions (since all relations in the semantic net are assumed to hold simultaneously) while the generalization language allows weak forms of disjunction and exceptions. Disjunction is introduced via the MAYBE annotation. This is equivalent to saying that a relation (unary or binary) holds OR it does not hold in the description. This is a restricted form of internal disjunction in which the set of permitted values (which are implicit in MAYBE annotation) are TRUE, FALSE, and DON´T KNOW. Exceptions are represented in the generalization language with the annotation NOT and MUST-NOT on arcs in the net. The generalization language uses variables to replace constants (e.g. object names) but makes no use of quantification[9].

The remainder of the evaluation of Winston´s learning system is characterized as teacher-dependant. Winston assumes that training instances are thoughtfully presented. The teacher must be cognizant of the current state, the goal state, and the internal induction mechanism in order to direct the learning process. By requiring that the teacher know the system´s knowledge state and plan the training sequence accordingly, Winston (largely) avoids certain complications:

- checking past instances -- The induction algorithm performs a depth-first search through the space of concept descriptions. A depth-first algorithm, in order to ensure consistency with past instances, must check past positive instances when specializing and past negative instances when generalizing. Winston does not perform these checks and thus cannot make strong claims of consistent concepts. By assuming that training instances differ from the current concept by only a few features (one feature for negative instances), induction proceeds in small steps. The generalization tends to "stay on track" and minor errors can be corrected with subsequent training instances.

[9]see [6] for a discussion of encoding quantification in networks.

- interference matching (combinatorial explosion) -- For each training instance presented, the induction algorithm finds the "best" match between the instance and the current concept description. The problem of multiple matchings is alleviated by assuming that the graphs being compared are highly similar.
- guaranteeing maximally-specific generalizations -- Positive instances must be presented in the "correct" order if maximally-specific generalizations are to be found. Only one candidate concept description is carried through the induction and generalized and specialized with instances. Permuting the order of the instances changes the final concept. In particular the concept may not be maximally-specific.

For the same reasons that Winston´s system is dependent on the teacher, it is also highly sensitive to noise in the training set.

3.2. Quinlan´s ID3 algorithm

The expressive power of ID3´s description languages is weak. Instances are represented with a feature vector and concepts are represented with a decision tree. The expressive power of a feature vector is minimal. External conjunction is implicit. Disjunction, exception, variables and quantification are not permitted. Decision trees allow both conjunction and disjunction of features. Variables and quantification are not used. This prevents natural encodings of structural and other n-ary relations. Exceptions, also, cannot be represented.

The only generalization rule used in forming the decision tree is the dropping condition rule. This is implicit in the operation of selecting a feature to sub-divide a set of instances. Some features are not used in the final decision tree because they are deemed non-criterial to the "minimal" concept description.

Balancing the weak expressive power of the description language and the modest generalization technique is ID3´s efficiency. Efficiency of both the induction process and the resulting decision tree can be measured:

- the induction algorithm -- ID3 was designed to perform induction over large sets of instances. Quinlan [19] states that computation time increases only linearly with difficulty as measured by the product of:

    * the number of given instances

    * the number of features used to describe instances

    * the number of nodes in the decision tree[10]

  A main source of complexity in the induction algorithm is evaluating an instance w.r.t. a feature. This operation must be performed $|F| \times |I|$ times at each node in the tree (where F is the set of features used by I, the set of instances). The efficiency of this step is dependent on the amount of useful knowledge encoded in the feature vector.

- the decision tree -- The final decision tree formed is minimal in that the <u>expected</u> classification time of an instance using the tree is no greater than the classification time of an "equivalent" tree.[11] This assumes that the set of instances "seen" by the induction algorithm to form a decision tree is representative of the distribution of the larger set of unseen instances.

ID3 requires no teacher assistance during the induction process. All training instances must be provided before processing begins. The induction proceeds depth-first and only one candidate concept description is maintained. Since all instances are present, the problem of checking past instances for consistency (required for incremental learning) is avoided. The concepts (rules) learned are maximally-general in the sense that (on average) the fewest number of features needed to classify instances for the concept are in the rule.

---

[10]this is exponential in the height of the tree.

[11]We say two decision trees DT_1 and DT_2 are equivalent if for all instances I, classifiable by either DT_1 or DT_2, classify(I,DT_1)=classify(I,DT_2).

As described in section 2-3, ID3 uses a beam search for efficient learning given a large set of training instances. Given this search strategy, we believe that ID3 can be made noise tolerant by relaxing the constraint that a decision tree be consistent with all instances, allowing a margin for error. Additional support for this belief is given by Quinlan [19] when describing the main findings for using a beam search. He found a correct and consistent tree was formed using only a small fraction of the total set of instances and the search was not sensitive to the size of the subset selected at each iteration. This indicates that a small margin for error will not disrupt learning in general. A consequence of the ID3 decision tree representation for rules is that the impact of small rule errors on final classification errors is a function of the position of the error in the decision tree. Rule errors near the root of the tree are more serious than errors near the leaves. Rules encoded as decision trees are analogous to rules encoded as production systems with an order imposed on the rule firing sequence.

3.3. Michalski et.al.´s INDUCE algorithm

As described in section 2.4, the description languages for INDUCE are as expressive as FOPC. Conjunction, disjunction, exception, variables and quantification are permitted. Moreover, there are syntactic modifications to FOPC which allow natural encodings. In addition, the generalization rules used are powerful. Constructive generalization adds descriptors to a generalization which are not present in the instances used in the generalization.

The INDUCE algorithm is non-incremental. The search technique has components of both a beam search and best-first search. A set of maximally-general candidate concept descriptions is found which is consistent with <u>all</u> negative instances and complete w.r.t. <u>one</u> positive instance. The best description (as defined by a domain-specific heuristic) is saved. The process repeats with another positive instance. The final rule is the disjunction of best descriptions for some (hopefully small) number of iterations.

INDUCE might demonstrate good noise immunity because of the structure of generalizations found. A generalization is basically a disjunction of terms where each term covers some subset of the positive instances. Noisy instances (assuming they are <u>very</u> noisy) are in separate terms which can be removed.

Clearly, the description languages are able to <u>represent</u> any concept for which FOPC is appropriate. But, guided by powerful generalization rules, what class of concepts can be <u>learned</u>? The problem encountered is the size of the search space. There are three points in the INDUCE algorithm in which non-determinism arises (refer to figure 2-9):

1. Expanding the set of candidates by applying inference rules (line 3) -- A non-deterministic selection from three powerful generalization rules with multiple bindings likely.

2. Generalizing consistent concept descriptions in attempt to make them complete (line 8) -- Again three generalization rules.

3. Implicit matching of instances with concept descriptions -- a pattern matcher is implicit which finds "best" match between an instance and a concept description. This is also used to determine if a concept covers an instance.

Countering this enormous search space, INDUCE employs a body of domain-specific heuristics to select the most promising candidate concept descriptions (lines 2,4, and 10 of figure 2-9). These pruning heuristics are essential and are applied at every opportunity during each iteration of the

search (formrule function performs one iteration). Surprisingly, INDUCE does not guide the selection of generalization rules with heuristics. INDUCE must select between five different rules to apply, each with the potential of multiple bindings to the current state.

However, as described by Lenat [11], heuristics have a limited domain of applicability. Outside of this domain, a heuristic can be useless or dangerous. We are unable to judge the <u>heuristic adequacy</u> [12] of the INDUCE algorithm. The "knowledge-intensive" approach is proving useful in problem solving, expert systems and natural language processing (among other areas) and should find application in inductive learning as well. However, evaluation of such a system is easily influenced by the quality and quantity of knowledge available to the system. This can conceal the domain-independant, formal properties of an algorithm.

3.4. Mitchell et.al.´s LEX program

LEX trades off the power of INDUCE, for example, for a guarantee of maximally specific, complete and consistent concept descriptions. The concept trees used for generalization and specialization combined with the frontiers of the bi-directional search implicitly define the version space. Also implicitly defined (reconstructable) are two sets of concept descriptions which LEX has ruled out. One set contains those descriptions found inconsistent with some past negative training instance. The second set contains those descriptions found incomplete with respect to some positive instance. LEX proceeds incrementally. For each new training instance, LEX reduces the size of the version space by moving a frontier (effectively moving candidate concept descriptions from the version space to one of the two sets of "ruled out" concepts). This ensures that only complete and consistent concept descriptions remain in the version space. Furthermore, we say that the frontiers are moved minimally with each training instance in the sense that only those candidates from the version space which are inconsistent with the current training instance are removed. This guarantees that LEX will find the maximally specific concept description.

The description languages and the generalization rules are central to the LEX version space approach. The description languages are essentially feature lists. The description languages allow more natural, domain specific, forms of description (e.g. mathematical notation for the domain of integration problems) but the expressive power is equivalent to a conjunction of features. The only generalization/specialization rule is climbing/descending concept hierarchy trees. By introducing more powerful rules, perhaps operating on a more expressive generalization language, LEX would forfeit the ability to re-construct the search space from the frontiers and to divide the search space into three sets: possible candidates, incomplete candidates, and inconsistent candidates.

LEX is capable of learning multiple-concepts. This is done by maintaining separate version spaces (search graphs) for each concept.

LEX differs from the other induction algorithms surveyed in the domain of application. Learning rules for problem solving (calculus problems for LEX) can be a dynamic process. For example, a problem solver can be incorporated with the induction algorithm or a heuristic function can measure proximity of a problem state to a goal.[12]

LEX takes advantage of the potential of learning dynamically with a teacherless four component system [17] (see figure 3-1).

- The problem generator -- this component proposes problems to be solved from which LEX can refine its knowledge base.
- The problem solver -- this component attempts to solve the proposed problem using the current knowledge.
- The critic -- this component analyses the resulting search graph. Credit assignment defines "appropriate" applications of a (partially-learned) rule from the knowledge base as those on the solution path. Inappropriate applications are those deviating from the solution path.
- The generalizer -- this component integrates the positive and negative instances of rule application (identified by the critic) with the current knowledge base. This corresponds to the version space induction algorithm.

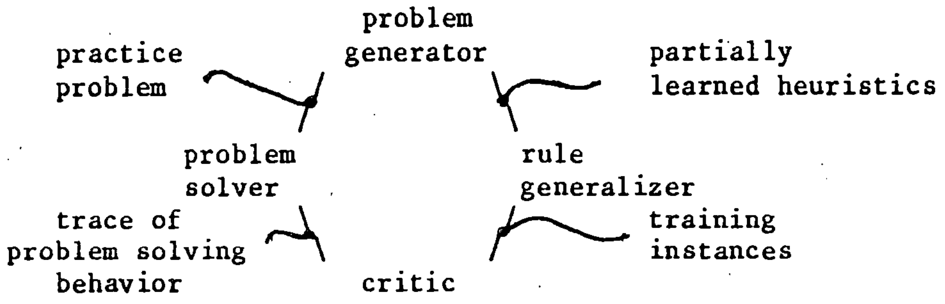


Figure 3-1: LEX´s Dynamic Learning Cycle (reproduced from [17], page 167)

One strong advantage of the version space algorithm is that LEX has a dynamic representation of what it knows. This meta-knowledge could be used by LEX´s problem generator. The problem generator could measure the degree of convergence on a concept (as measured by the distance between the frontiers of

[12]This property of learning in dynamic domains was used in building a system for learning rules for solving simultaneous linear equations. See [8, 9, 10].

the search graph). A problem could be proposed which bisects the version space, for an optimum learning rate. In fact, the meta-knowledge is not currently used by the problem generator, although Mitchell documents its potential [17]. Another use of this meta-knowledge allows LEX to measure the confidence of a partially-learned concept. This could allow the problem solver and the critic to prudently use partially-learned concepts when expanding a search graph or performing credit assignment. A third use of the meta-knowledge is for learning with noisy training sets. Mitchell [15, 2] has defined a modified version space algorithm which maintains multiple boundary sets. In the modified algorithm, the sets S_0 and G_0 correspond to the sets S and G in the noise-free algorithm. Added to the storage requirements are S_1-S_n and G_1-G_n where each description in the set S_i is consistent with all but i of the positive training instances and each description in the set G_i is consistent with all but i of the negative instances, for i between 0 and n. The algorithm detects when a pair of boundaries S_i and G_i cross. The algorithm concludes that at least i instances were noisy and looks for convergence on a concept bounded by S_{i+1} and G_{i+1}.

Young et.al. [25, 1] have devised a space-saving modification to the version space algorithm. Basically, they propose eliminating the explicit representation of the frontiers by an implicit representation. Markers are placed at nodes in each concept tree corresponding to the most specific and most general abstractions of the each concept which is consistent with past instances. This is analogous to distributing the frontiers from the concept version space to each of the concept trees involved in the total concept.

4. Conclusions

We have surveyed the induction algorithms used by four significant learning systems. These systems are in the class of learning by examples. We have detailed the control mechanism and the knowledge representation used by each. Winston´s ARCH system was found to be domain and teacher dependant. By relying on near examples and near misses in the training set, ARCH avoids incorrect concepts and combinatorial explosion. Quinlan´s ID3 algorithm buys efficiency and simplicity at the expense of expressive power. Michalski, et.al.´s INDUCE program has the expressive power of FOPC, and powerful generalization rules. It represents a significant effort to apply more domain knowledge to the learning process. Finally, Mitchell, et.al.´s LEX system guarantees complete and consistent concept descriptions, but employs weak description languages and generalization rules. We have attempted to illuminate the commonalities and differences of these algorithms. More important, however, are the ubiquitous trade-offs of expressive power versus efficiency, domain independence versus strong methods, and teacher guidance versus combinatorial explosion, that characterize the state of the art of machine learning.

REFERENCES

1. Bundy, A. and Silver, B. A Critical Survey of Rule Learning Programs. 169, University of Edinburgh, Dept. of AI, 1981.

2. Cohen, P.R. and Feigenbaum, E.A. The Handbook of Artificial Intelligence. William Kaufman, 1982. chapter 14

3. Deliyanni, A. and Kowalski, R.A. Logic and Semantic Networks. ACM 22 (1979), 184-192.

4. Dietterich, T.G. and Michalski, R.S. A Comparative Review of Selected Methods for Learning From Examples. In Michalski,R.S., Carbonell,J.G., Mitchell,T.M., Ed., Machine Learning, Tiogo Publishing, 1983.

5. Hayes-Roth, F., and McDermott, J. An Interference Matching Technique for Inducing Abstractions. Communications of the ACM 21 (1978), 401-410.

6. Hendrix, G.G. Encoding Knowledge in Partitioned Networks. In Findler, N.V., Ed., Associative Networks - The Representation and Use of Knowledge in Computers, Academic Press, 1979, pp. 51-92.

7. Hunt,E.B., Marin,J. and Stone, P.T. Experiments in Induction. Academic Press, 1966.

8. Kibler, D.F. and Porter, B.W. Episodic Learning. 194, University of California, Irvine, 1983.

9. Kibler, D.F. and Porter, B.W. Perturbation: A Means for Guiding Generalization. IJCAI (1983), [to appear].

10. Kibler, D.F. and Porter, B.W. Episodic Learning. AAAI (1983), [to appear].

11. Lenat, D.B. The Role of Heuristics in Learning by Discovery: Three Case Studies. In Michalski,R.S., Carbonell,J.G., Mitchell,T.M., Ed., Machine Learning, Tiogo Publishing, 1983.

12. McCarthy, J. and Hayes, P.J. Some Philosophical Problems from the Standpoint of Artificial Intelligence. Machine Intelligence 4 (1969), 463-502.

13. Michalski,R.S., Carbonell,J.G., Mitchell,T.M. Machine Learning. Tiogo Publishing, 1983.

14. Michalski, R.S. A Theory and Methodology of Inductive Learning. In Michalski,R.S., Carbonell,J.G., Mitchell,T.M., Ed., Machine Learning, Tiogo Publishing, 1983.

15. Mitchell, T.M. Version Spaces: An approach to Concept Learning. Ph.D. Th., Stanford, 1978. TR STAN-CS-78-711

16. Mitchell, T.M. Generalization as Search. Artificial Intelligence 18 (1982), 203-226.

17. Mitchell, T.M., Utgoff, P.E., Nudel, B, and Banerji, R. Learning by Experimentation: Acquiring and Refining Problem-Solving Heuistics. In Michalski,R.S., Carbonell,J.G., Mitchell,T.M., Ed., Machine Learning, Tiogo Publishing, 1983.

18. Quinlan, J.R. Induction over Large Data Bases. HPP-79-14, Heuristic Programming Project, Stanford University, 1979.

19. Quinlan, J.R. Learning Efficient Classification Procedures and their Application to Chess End Games. In Michalski,R.S., Carbonell,J.G., Mitchell,T.M., Ed., Machine Learning, Tiogo Publishing, 1983.

20. Sammut, C. Concept Learning by Experiment. IJCAI 7 (1981), 104-105.

21. Soloway, E.M. Learning Interpretation + Generalization: A Case Study in Knowledge-Directed Learning. Ph.D. Th., University of Massachusetts at Amherst, 1978.

22. Vere, S.A. Induction of concepts in the predicate calculus. IJCAI 4 (1975), 281-287.

23. Winston, P.H. Learning structural description from examples. In Winston, P.H., Ed., The Psychology of Computer Vision, McGraw-Hill, 1975.

24. Winston, P.H. Artificial Intelligence. Addison-Wesley, 1977.

25. Young, R.M., Plotkin, G.D. and Linz, R.F. Analysis of an Extended Concept-Learning Task. IJCAI 5 (1977), 285.