Pac-Learning a Restricted Class of Recursive Logic Programs

William W. Cohen
AT&T Bell Laboratories
600 Mountain Avenue Murray Hill, NJ 07974
wcohen@research.att.com

Abstract

A crucial problem in "inductive logic programming" is learning recursive logic programs from examples alone; current systems such as GOLEM and FOIL often achieve success only for carefully selected sets of examples. We describe a program called FORCE2 that uses the new technique of "forced simulation" to learn two-clause "closed" linear recursive ij-determinate programs; although this class of programs is fairly restricted, it does include most of the standard benchmark problems. Experimentally, FORCE2 requires fewer examples than FOIL, and is more accurate when learning from randomly chosen datasets. Formally, FORCE2 is also shown to be a pac-learning algorithm in a variant of Valiant's [1984] model, in which we assume the ability to make two queries: one which gives an upper bound on the depth of the proof for an example, and one which determines if an example can be proved in unit depth.

Introduction

An increasingly active area of research in machine learning is *inductive logic programming (ILP)*. ILP systems, like conventional concept learning systems, typically learn classification knowledge from a set of randomly chosen examples; however, ILP systems use *logic programs*—typically some subset of function-free Prolog—to represent this learned knowledge. This representation can be much more expressive than representations (such as decision trees) that are based on propositional logic; for example, Quinlan [1990] has shown that FOIL can learn the transitive closure of a directed acyclic graph, a concept that cannot be expressed in propositional logic.

A crucial problem in ILP is that of learning *recursive* logic programs from examples alone. Early work in ILP describes a number of systems that learn recursive programs, and some of them have convergence proofs [Shapiro, 1982; Banerji, 1988; Muggleton and Buntine, 1988]; however, these systems rely on fairly powerful queries (e.g. membership and subset queries) to achieve these results. More recent systems, such as FOIL [Quinlan, 1990] and GOLEM [Muggleton and Feng, 1992] have been experimentally successful in learning recursive programs from examples alone. While the results obtained with these systems are impressive, they are limited in two ways.

Non-random samples. In most published experiments in which recursive concepts are learned, the samples used in learning are not randomly selected examples of the target concept; instead, the samples are carefully constructed. For example, in using FOIL to learn the recursive concept *member*, Quinlan gives as examples all membership relations over the list $[a, b, [c]]$ and its substructures; providing examples for all substructures makes it relatively easy for FOIL's information gain metric to estimate the usefulness of a recursive call. It is unclear to what extent recursive programs can be learned from a random, unprepared sample.

Lack of formal justification. Although parts of the GOLEM algorithm have been carefully analyzed, both FOIL and GOLEM make use of a number of heuristics, which makes them quite difficult to analyze. Ideally, one would like rigorous formal justification for an algorithm for learning recursive concepts, as well as experimental results on problems of practical interest.[1]

In this paper, we will describe a new algorithm called FORCE2 for learning a restricted class of recursive logic programs: namely, the class of two-clause "closed" linear recursive ij-determinate programs. This class is fairly restricted, but does include many of the standard benchmark problems, including list reversal, list append, and integer multiply. FORCE2 uses a new technique called *forced simulation* to choose the recursive call in the learned program. Formally, FORCE2 will be shown to be a *pac-learning* algorithm for this class; hence it learns against any distribution of examples in polytime, and does not require a hand-prepared sample. Experimentally, FORCE2 requires fewer examples than FOIL, has much less difficulty with samples that have not been prepared, and

[1]Recently, the pac-learnability of the class of programs learned by GOLEM has been studied; however, for recursive programs the analysis assumed membership and subset queries [Džeroski *et al.*, 1992].

is fast enough to be practical.

Although FORCE2 makes no queries *per se*, it does require a certain amount of extra information. In particular, FORCE2 must be given both a characterization of the "depth complexity" of the program to be learned, and also a procedure for determining when an instance is an example of the base case of the recursion. For example, to learn the textbook definition of *append*, one might supply the following

MAXDEPTH(append(Xs,Ys,Zs)) $\equiv$
    length(Xs)+1.
BASECASE(append(Xs,Ys,Zs)) $\equiv$
    if null(Xs) then **true** else **false**

The user need give only an upper bound on the depth complexity, not a precise bound, and need give only sufficient (not necessary and sufficient) conditions for membership in the base case. In Sections and we will discuss, from both a formal and practical viewpoint, the degree to which this extra information is needed: there are circumstances under which both the depth bound and the base-case characterization can be dispensed with. First, however, we will describe the class of logic programs that FORCE2 learns, and present the FORCE2 algorithm and its accompanying analysis.

The class of learnable programs

In a typical ILP system, the user will provide both a set of examples and a *background theory* K: the learning system must then find a logic program P such that $P \wedge K \vdash e^+$ for every positive example e^+, and $P \wedge K \not\vdash e^-$ for every negative example e^-. As a concrete example, if the target concept is a function-free version of the usual definition of *append*, the user might provide a background theory K defining the predicate *null(A)* to be true when A is the empty list, and defining the predicate *components(A,B,C)* to be true when A is a list with head B and tail C. The learned program P can then use these predicates: for example, P might be the program

```
append(Xs,Ys,Ys) ←
    null(Xs).
append(Xs,Ys,Zs) ←
    components(Xs,X,Xs1),
    components(Zs,X,Zs1),
    append(Xs1,Ys,Zs1).
```

FORCE2, like both GOLEM and FOIL, requires the background theory K to be a set of ground unit clauses (aka relations, a set of atomic facts, or a model.) Like FOIL, our implementation of FORCE2 also assumes that the program to be learned contains no function symbols; however this restriction is not necessary for our formal results.

The actual programs learned by the FORCE2 procedure must satisfy four additional restrictions. First, a learned program must contain *exactly two clauses:* one recursive clause, and one clause representing the base case for the recursion. Second, the recursive clause must be *linearly recursive:* that is, the body of the clause must contain only a single "recursive literal", where a recursive literal is one with the same principle functor as the head of the clause. Third, the recursive literal must be *closed*: a literal is closed if it has no "output variables".[2] Finally, the program must contain no function symbols, and must be *ij-determinate*. The condition of ij-determinacy was first used in the GOLEM system, and variations of it have subsequently been adopted by FOIL [Quinlan, 1991], LINUS [Lavrač and Džeroski, 1992], and GRENDEL [Cohen, 1993b]. It is defined in detail by Muggleton and Feng [1992].[3]

The FORCE2 learning algorithm

The FORCE2 algorithm is summarized in Figure 1. The algorithm has two explicit inputs: a set of positive examples S^+, and a set of negative examples S^-. There are also some additional inputs that (for readability) we have made implicit: the user also provides a function MAXDEPTH(e) that returns an upper bound on the depth complexity of the target program, a function BASECASE(e) that returns "true" whenever e is an example of the base clause, a background theory K (defining only predicates of arity j or less), and a depth bound i. The output of FORCE2 is a two-clause linear and closed recursive ij-determinate logic program.

The FORCE2 algorithm takes advantage of an important property of ij-determinate clauses: for such clauses, the *relative least general generalization*[4] *(rlgg)* of a set of examples is of size polynomial in the number of predicates defined in K, is unique, and can be tractably computed. FORCE2 actually uses two rlgg operators: one that finds the least general clause that covers a set of examples, and one that takes an initial clause C_0 and a single example e and returns the least general clause $C_1 \supseteq C_0$ that covers e.

The first step of the algorithm is to use the BASECASE function to split the positive examples into two

[2] If B_i is a literal of the (ordered) Horn clause $A \leftarrow B_1, \ldots, B_l$, then the *input variables* of the literal B_i are those variables appearing in B_i that also appear in the clause $A \leftarrow B_1, \ldots, B_{i-1}$; all other variables appearing in B_i are called *output variables*.

[3] Briefly, a literal B_i is *determinate* if its output variables have at most one possible binding, given the binding of the input variables. If a variable V appears in the head of a clause, then the depth of V is zero, and otherwise, if B_i is the first literal containing the variable V and d is the maximal depth of the input variables of B_i, the depth of V is $d+1$. Finally, a clause is *ij-determinate* if its body contains only literals of arity j or less, if all literals in the body are determinate, and if the clause contains only variables of depth less than or equal to i. Muggleton and Feng argue that many common recursive logic programs are ij-determinate for small i and j.

[4] The rlgg of a set of examples S (with respect a background theory K) is the least general clause C such that $\forall e \in S,\ C \wedge K \vdash e$.

```
program FORCE2(S+, S-)
  C_base := rlgg({e ∈ S+ : BASECASE(e)})
  C_rec := rlgg({e ∈ S+ : ¬BASECASE(e)})
  for each recursive literal L_r over variables(C_rec) do
    for each e+ ∈ S+ do
      force-sim(e+, C_base, C_rec, L_r)
      P := the logic program containing the
       clauses C_base and A ← B_1, ..., B_l, L_r,
       where C_rec = A ← B_1, ..., B_l
      if an error was signaled in force-sim then
        reset C_base and C_rec to their original values
        break from inner "for" loop and try the next L_r
      endif
    endfor
    if P consistent with S- then return P
  endfor
  return "no consistent program"
end
```

```
subroutine force-sim(e, C_base, C_rec, L_r)
  if BASECASE(e) then
    C_base := rlgg(C_base, e)
  else
    C_rec := rlgg(C_rec, e)
    if any variable in L_r is not in the new C_rec or
     force-sim has recursed more than MAXDEPTH(e_0)
     times, where e_0 is the example e used in the
     top-level call of force-sim
    then
      signal an error and exit
    else
      A := the head of C_rec
      B_1, ..., B_l := the body of C_rec
      e' := L_r θ, where θ is such that Aθ = e
       and K ⊢ B_1 θ, ..., B_l θ
      force-sim(e', K, C_base, C_rec, L_r)
    endif
  endif
end
```

Figure 1: The FORCE2 algorithm

sets: the examples of the base clause, and the examples of the recursive clause. The rlggs C_{base} and C_{rec} of these two sets of examples are then computed; informally, these rlggs will be used as initial guesses at the base clause and the recursive clause respectively. (More accurately, C_{rec} is a guess at the *non-recursive portion* of the recursive clause being learned.) As an example, we used FORCE2 to learn the *append* program (given as an example on page) from a small set of examples; the BASECASE and MAXDEPTH functions were as given on page . The rlgg of the three positive examples of the base case was the following:[5]

C_{base}: append(A,B,C) ←
    components(B,D,E),
    null(A),
    B=C.

The rlgg of the non-base case positive examples was the following:

C_{rec}: append(A,B,C) ←
    components(A,D,E),
    components(C,F,G),
    D=F.

Notice that C_{base} is over-specific, since it requires B to be a non-empty list, and also that C_{rec} does *not* contain a recursive call. The remaining steps of the algorithm are designed to find an appropriate recursive call.

From the way in which C_{base} and C_{rec} were constructed, one might suspect that adding a recursive literal L_r to C_{rec} would yield the least general program (in our class) that uses that recursive call and is consistent with the positive data; however, this is false. Consider adding the (correct) recursive call $L_r = append(E, B, G)$ to C_{rec}, to yield the program

append(A,B,C) ←
    components(B,D,E),
    null(A),
    B=C.
append(A,B,C) ←
    components(A,D,E),
    components(C,F,G),
    D=F,
    append(E,B,G).

Now consider the non-basecase positive example $e = append([2], [], [2])$. Example e is covered by C_{rec}, but not by the program above; the problem is the the subgoal $e' = append([],[],[])$ is not covered by the base clause.

The subroutine *force-sim* handles this problem. Conceptually, it will simulate the hypothesized logic program on e, except that when the logic program would fail on e or any subgoal of e, the rlgg operator is used to generalize the program so that it will succeed. The effect is to construct the least general program with the recursive call L_r that covers e.

We will illustrate this crucial subroutine by example. Suppose FORCE2 has chosen the (correct) recursive literal $L_r = append(E,B,G)$, using the C_{base} and

[5] As the examples show, our algorithm for constructing lgg's is a bit nonstandard. A distinct variable is placed in every position that could contain an output variable, and equalities are then expressed by explicitly adding equality literals (like $A{=}C$ and $D{=}F$ in the examples above.) This encoding for an lgg is not the most compact one; however it is only polynomially larger than necessary, and simplifies the implementation. With the usual encoding the enumeration of recursive literals must be interleaved with the *force-sim* routine.

C_{rec} given above, and consider forcibly simulating the positive example *append([1,2],[],[1,2])*. The subroutine *force-sim* determines that e is not a BASECASE; thus it will generalize C_{rec} to cover e by replacing C_{rec} with the rlgg of C_{rec} and e; in this case C_{rec} is unchanged. The next step is to continue the simulation by determining what subgoal would be generated by the proposed recursive call; in this case, the recursive call *append(E,B,F)* would generate the subgoal e' =*append([2],[],[2])*. The subroutine *force-sim* is then called on e', and determines that is not a BASECASE; again, replacing C_{rec} with its rlgg against e' leaves C_{rec} unchanged. Finally, the routine again computes the recursive subgoal e'' =*append([],[],[])*, and recursively calls *force-sim* on e''. It determines that e'' is a BASECASE, and generalizes C_{base} to cover it, again using the rlgg operator. The final result is that C_{rec} is unchanged, and that C_{base} has been generalized to

append(A,B,C) ← null(A),B=C.

This is the least generalization of the initial program (in the class of closed linear recursive two-clause programs) that covers the example *append([1,2],[],[1,2])*.

For an incorrect choice of L_r, forced simulation may fail: for example, if FORCE2 were testing the incorrect recursive literal L_r =*append(A,A,C)*, then given the example *append([1,2],[],[1,2]) force-sim* would loop, repeatedly generating the same subgoal $e' = e'' = e''' \ldots$. This loop would be detected when the depth bound was violated, and an error would be signaled, indicating that no valid generalization of the program covers the example. For incorrect but non-looping recursive literals, forced simulation may lead to an overgeneral hypothesis: for example, given the incorrect recursive literal L_r =*append(B,A,E)* and the example *append([1,2],[],[1,2]) force-sim* would subgoal to e' =*append([],[1,2],[2])*, and generalize C_{base} to the clause *append(A,B,C) ← null(A)*. With sufficient negative examples, this overgenerality would be detected.

The remainder of the algorithm is straightforward. The inner **for** loop of the FORCE2 algorithm uses repeated forced simulations to find a least general hypothesis that covers all the positive examples, given a particular choice for a recursive literal L_r; if this least general hypothesis exists and is consistent with the negative examples, it will be returned as the hypothesis of the learner. The outer **for** loop exhaustively tests all of the possible recursive literals; it is not hard to see that for a fixed maximal arity j there are polynomially many possible L_r's.[6]

[6] Since C_{rec} is of polynomial size, it contains polynomially many variables. Let $p(n)$ be the number of variables in C_{rec} and t be the arity of the target predicate; there are at most $p(n)^t \leq p(n)^j$ recursive literals.

Formal results

Formalizing the preceding discussion leads to the following theorem.

Theorem 1 *If the training data is labeled according to some two-clause linear and closed recursive ij-determinate logic program, then FORCE2 will output a least general hypothesis consistent with the training data. Furthermore, assuming that that MAXDEPTH and BASECASE can be computed in polynomial time, for any fixed i and j, FORCE2 runs in time polynomial in the number of predicates defined in K, the number of training examples, and the largest value of MAXDEPTH for any training example.*

It is known that ij-determinate clauses are of size polynomial in the number of background predicates; this implies that the VC-dimension [Blumer *et al.*, 1986] of the class of 2-clause ij-determinate programs over a background theory K is polynomial in the size of K. Thus an immediate consequence of the preceding theorem is the following:

Corollary 1 *For any background theory K, FORCE2 is a pac-learning algorithm [Valiant, 1984] for the concept class of two-clause linear and closed recursive ij-determinate logic programs over K.*

To our knowledge, this is the first result showing the polynomial learnability of any class of recursive programs in a learning model disallowing queries.

It might be argued that the MAXDEPTH and BASECASE functions act as "pseudo-queries", and hence that our learning model is not truly passive. It should be noted, however, that the MAXDEPTH bound is not needed for the formal results, as the constant depth bound of $(j|K|)^j$ is sufficient to detect looping. (In practice, however, learning is much faster with an appropriate depth bound.) The BASECASE function is harder to dispense with, but even here a positive formal result is obtainable. If one assumes that the base clause is part of K, it is reasonable to consider the learnability of the class of *one-clause* linear and closed recursive ij-determinate logic programs over K. It can be shown that this class is pac-learnable, using a slightly modified version of FORCE2 that never changes the base clause.

This pac-learnability result can also be strengthened in a number of technical ways. A variant of the FORCE2 algorithm can be shown to achieve exact identification from equivalence queries; this learning criterion is strictly stronger than pac-learnability [Blum, 1990]. Another extension is suggested by the fact that FORCE2 returns a program that is maximally specific, given a particular choice of recursive call. Instead of returning a single program, one could enumerate all (polynomially many) consistent least programs; this could be used to tractably encode the version space of all consistent programs using Hirsh's $[S, N]$ representation for version spaces [1992].

Table 1: FORCE2 *vs.* FOIL on subsets of a good sample of *multiply*

%full	FOIL error	FORCE2 error	(cpu)	%full	FOIL error	FORCE2 error	(cpu)
2	6.10%	2.60%	(20)	40	1.70%	0.00%	(121)
5	5.10%	1.10%	(21)	60	0.84%	0.00%	(167)
10	4.80%	0.73%	(33)	80	0.19%	0.00%	(216)
20	3.00%	0.14%	(63)	100	0.00%	0.00%	(294)

Table 2: FORCE2 *vs.* FOIL on random samples

	"natural" samples		variant distributions				
Problem	FOIL	FORCE2	comment	FOIL	FORCE2	Straw1	Straw2
append	1.7%	0.0%	near miss	27.7%	0.0%	27.3%	40.3%
heapsort	3.7%	0.0%	near miss	7.5%	0.0%	19.7%	0.0%
member	13.4%	0.0%	length< 10	15.1%	0.0%	41.1%	0.0%
reverse1	5.4%	0.0%	near miss	13.9%	0.0%	8.4%	8.0%

Experimental results

To further evaluate FORCE2, the algorithm was implemented and compared to FOIL4, the most recent implementation of FOIL. The first experiment we performed used the *multiply* problem, which is included with Quinlan's distribution FOIL4.[7] We presented progressively larger random subsets of the full dataset to FOIL and FORCE2 and estimated their error rates using the usual crossvalidation techniques. The results, shown in Table 1, show that FORCE2 generalizes more quickly than FOIL. For this dataset, guessing the most prevalent class gives about a 4.5% error rate; thus FOIL's performance for datasets less than 20% complete is actually quite poor.

CPU times[8] are also given for FORCE2, showing that FORCE2's run time scales linearly with the number of examples. No systematic time comparison with FOIL4 was attempted, as FORCE2 is implemented in Prolog, and FOIL4 in C. However we noted that even though FORCE2 is about 30 times slower than FOIL4 on the full hand-prepared dataset (294 seconds to 9 seconds) their speeds are were comparable on random subsamples of *multiply*: overall, FORCE2 averaged 91.6 seconds a run, and FOIL4 averaged 88.6 seconds a run. Run times for FORCE2 on other benchmarks (reported below) were also comparable: FORCE2 averaged 7 seconds for problems in Table 2 to FOIL4's 10.9 seconds.

Table 2 compares FORCE2 to FOIL on a number of other benchmarks, using training sets of 100 examples; the intent of this experiment was to evaluate performance on randomly-selected samples, and to determine how sensitive performance is to the distribution. First we presented FOIL and FORCE2 with samples generated by what seemed to us the most natural random procedures.[9] These results are shown in the left-hand column. Next, we compared FORCE2 and FOIL on some harder variations of the "natural" distribution;[10] these results are shown in the right-hand column. FOIL's performance degraded noticeably on these distributions, but FORCE2's was unchanged.

There are (at least) two possible explanations for these results. First, FORCE2 has a more restricted bias for FOIL; thus the well-known generality-power tradeoff would predict that FORCE2 would outperform FOIL. Second, FORCE2 uses a more sophisticated method of choosing recursive literals than FOIL. FORCE2 evaluates a recursive literal by using forced simulation of the positive examples, followed by a consistency test against the negative examples. FOIL uses

[7] The distributed dataset contains 1056 examples: all positive examples of *mult(X,Y,Z)* where X and Y are both less than seven, and all negative examples *mult(X,Y,Z)* where X, Y, and Z appear as arguments to some positive example. All results are averaged over five trials.

[8] In seconds on a Sparc 1+.

[9] For *member*, a random list was generated by choosing a number L uniformly between one and four, then building a random list of length L over the atoms $a,\ldots,z$; positive examples were constructed by choosing a random element of the list, and negative examples by choosing a random non-element of the list. For *append*, positive examples were generated by choosing two lists at random (here allowing null lists) as the first two arguments, and negative examples by choosing three lists at random. Examples for *reverse1* and *heapsort* were generated a similar way; *reverse1* is a naive reverse program where the background theory defines a predicate that appends a single element to the tail of a list. Equal numbers of positive and negative examples were generated for all problems, and error rates were estimated using 1000 examples generated from the same distribution.

[10] On *member*, we increased the maximum length of a list from five to ten. On the remaining problems, we generated negative examples by taking a random positive example and randomly changing one element of the final argument; these "near miss" examples provide more information, but are harder to distinguish from the positive examples.

information gain (a heuristic metric based on coverage of the positive and negative examples) to choose all literals, including recursive ones. Application of any coverage-based metric to recursive literals is not straightforward: in general one does not know if a recursive call to a predicate P will succeed or fail, because the full definition for P has not been learned. To circumvent this problem, FOIL uses the examples of P as an oracle for the predicate P: a subgoal of P is considered to succeed when there is a positive example that unifies with the subgoal.

A disadvantage of FOIL's method is that for random samples, the dataset can be a rather noisy oracle. In addition to making it harder to choose the right recursive literals, this means that FOIL's assessment of which examples are covered by a clause may be inaccurate, causing FOIL to learn redundant or inaccurate clauses.

To clarify the reasons for the difference in performance, we performed a final study in which we ran two "strawmen" learning algorithms on the harder distributions. *Straw1* outputs FORCE2's initial nonrecursive hypothesis without adding any recursive literal; *Straw2* adds a single closed recursive literal chosen using the information gain metric.

Our study indicates that both explanations are partially correct. On the *member* and *heapsort* problems, the good performance of Straw2 indicates that information gain is sufficient to choose the correct recursive literal, and hence the primary reason for FORCE2's superiority is its more restrictive bias.[11] However, for *reverse1* and *append*, even the strongly-biased Straw2 does poorly, indicating that on these problems information gain is not effective at choosing an accurate recursive literal; for these problems, particularly *append*, FOIL's broader bias may actually be helpful.[12]

Concluding remarks

This paper has addressed the problem of learning recursive logic programs from examples alone against an arbitrary distribution. We presented FORCE2, a procedure that learns the restricted class of two-clause linear and closed recursive ij-determinate programs; this class includes many of the standard benchmark problems. Experimentally, FORCE2 requires fewer examples than FOIL, and is less sensitive to the distribution of examples.

More importantly, FORCE2 can be proved to pac-learn any program in this class. This result is surprising, as previous positive results have either considered only nonrecursive concepts (e.g. [Page and Frisch, 1992]) or have assumed the ability to make membership or subset queries (e.g. [Shapiro, 1982; Banerji, 1988; Džeroski *et al.*, 1992]). We make use of two additional sources of information, namely the BASECASE and MAXDEPTH functions; however only one of these (the BASECASE function) is necessary for our formal results. Also, as noted in the discussion following Theorem 1 a positive result can be obtained for a slightly more restricted language from labeled examples alone.

There are a number of possible extensions to the FORCE2 algorithm. Although the BASECASE function is formally necessary for pac-learning,[13] it is probably unnecessary in practice, given a distribution that provides a reasonable number of instances of the base case. Learners like GOLEM and FOIL can learn nonrecursive clauses relatively easily; thus one might first learn the base case(s), and then use a slightly modified version of FORCE2 to learn the recursive clause.[14] Such a learner would be more useful, albeit harder to analyze.

It would also be highly desirable to extend the class of learnable programs. FORCE2 can be easily modified to learn 2-clause programs with k closed recursive calls: one simply enumerates all k-tuples of recursive literals $L_{r_1}, \ldots, L_{r_k}$, and modifies *force-sim* to subgoal from an example to the appropriate k subgoals $e'_1, \ldots, e'_k$. This learning algorithm generates a consistent clause, but can take time exponential in the depth bound given in MAXDEPTH and hence is suitable only for problems with a logarithmic depth bound.[15] Although space does not permit a full explanation, it is also possible to extend FORCE2 to learn non-closed 2-clause recursive programs.

On the other hand, several hardness results are known if the ij-determinacy condition is relaxed, or if less limited recursion is allowed; for example, learning a 2-clause ij-determinate program with a polynomial number of recursive calls is cryptographically hard, even if the base case for the recursion is known [Cohen, 1993a]. In another recent paper [Cohen, 1993c] we also show that learning an ij-determinate program with a polynomial number of linear recursive clauses is cryptographically hard. However, the learnability

[11] Examining traces of FOIL4 on these problems also supports this view. For *member*, FOIL4 learned the correct program twice and non-recursive approximations with high error rates three times; due to the "noisy oracle problem" FOIL4 believes the correct program to have an error rate of more than 30%, so has no good reason for preferring it. On the *heapsort* problem FOIL4 typically learns several redundant clauses. Thus, for both problems, a bias toward two-clause recursive programs would be beneficial.

[12] Unlike Straw2, FOIL has the option of learning a nonrecursive approximation of the target concept, rather than making a forced choice among recursive literals based on inadequate statistics.

[13] Without it, learning 2-clause ij-determinate programs is as hard as learning 2-term DNF, which is known to be intractable [Kearns *et al.*, 1987].

[14] The modification is to disallow changes to the base clause(s).

[15] This is unsurprising, since the time complexity of the learned program can be exponential in its depth complexity.

of ij-determinate programs with a constant number of clauses, each with a constant number of recursive calls, remains open.

References

(Banerji, 1988) Ranan Banerji. Learning theories in a subset of polyadic logic. In *Proceedings of the 1988 Workshop on Computational Learning Theory*, Boston, Massachusetts, 1988.

(Blum, 1990) Avrim Blum. Separating PAC and mistake-bound learning models over the boolean domain. In *Proceedings of the Third Annual Workshop on Computational Learning Theory*, Rochester, New York, 1990. Morgan Kaufmann.

(Blumer *et al.*, 1986) Anselm Blumer, Andrezj Ehrenfeucht, David Haussler, and Manfred Warmuth. Classifying learnable concepts with the Vapnik-Chervonenkis dimension. In *18th Annual Symposium on the Theory of Computing*. ACM Press, 1986.

(Cohen, 1993a) William Cohen. Cryptographic limitations on learning one-clause logic programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, Washington, D.C., 1993. To appear.

(Cohen, 1993b) William Cohen. Explicit biases and inductive logic programming. In preparation, 1993.

(Cohen, 1993c) William Cohen. Learnability of restricted logic programs. Submitted to ILP-93, 1993.

(Džeroski *et al.*, 1992) Savso Džeroski, Stephen Muggleton, and Stuart Russell. Pac-learnability of determinate logic programs. In *Proceedings of the 1992 Workshop on Computational Learning Theory*, Pittsburgh, Pennsylvania, 1992.

(Hirsh, 1992) Haym Hirsh. Polynomial-time learning with version spaces. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, California, 1992. MIT Press.

(Kearns *et al.*, 1987) Micheal Kearns, Ming Li, Leonard Pitt, and Les Valiant. On the learnability of boolean formulae. In *19th Annual Symposium on the Theory of Computing*. ACM Press, 1987.

(Lavrač and Džeroski, 1992) Nada Lavrač and Sašo Džeroski. Background knowledge and declarative bias in inductive concept learning. In K. P. Jantke, editor, *Analogical and Inductive Inference: International Workshop AII'92*. Springer Verlag, Daghstuhl Castle, Germany, 1992. Lecture in Artificial Intelligence Series #642.

(Muggleton and Buntine, 1988) Steven Muggleton and Wray Buntine. Machine invention of first order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, Ann Arbor, Michigan, 1988. Morgan Kaufmann.

(Muggleton and Feng, 1992) Steven Muggleton and Cao Feng. Efficient induction of logic programs. In *Inductive Logic Programming*. Academic Press, 1992.

(Page and Frisch, 1992) C. D. Page and A. M. Frisch. Generalization and learnability: A study of constrained atoms. In *Inductive Logic Programming*. Academic Press, 1992.

(Quinlan, 1990) J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3), 1990.

(Quinlan, 1991) J. Ross Quinlan. Determinate literals in inductive logic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, Ithaca, New York, 1991. Morgan Kaufmann.

(Shapiro, 1982) Ehud Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.

(Valiant, 1984) L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11), November 1984.