PAC-Learnability of Determinate Logic Programs

Sašo Džeroski, Stephen Muggleton, Stuart Russell
The Turing Institute Limited, 36 North Hanover Street, Glasgow G1 2AD, Scotland, UK

Address for correspondence:
Sašo Džeroski, Institut Jožef Stefan, Jamova 39, 61111 Ljubljana, Slovenia
email: saso@ijs.yu, saso@turing.ac.uk

Abstract

The field of *Inductive Logic Programming* (ILP) is concerned with inducing logic programs from examples in the presence of background knowledge. This paper defines the ILP problem, and describes the various syntactic restrictions that are commonly used for learning first-order representations. We then derive some positive results concerning the learnability of these restricted classes of logic programs, by reducing the ILP problem to a standard propositional learning problem. More specifically, k-clause predicate definitions consisting of determinate, function-free, non-recursive Horn clauses with variables of bounded depth are polynomially learnable under a broad class of probability distributions, called simple distributions. Similarly, recursive k-clause definitions are polynomially learnable under simple distributions if we allow existential and membership queries about the target concept.

1 Introduction

Most successes within the field of machine learning have derived from systems that construct hypotheses within propositional logic, and, unsurprisingly, PAC learning theory has been applied principally to the learnability of subsets of propositional concept definitions. As relatively few concept classes are polynomially learnable under arbitrary probability distributions (for example, k-CNF, k-DNF [Valiant 1984, Haussler 1988], and k-DL [Rivest 1987]), learnability under specific distributions or classes of distributions has also been studied [Benedek and Itai 1988].

Recently, Li and Vitányi [Li and Vitanyi 1991] have shown that several interesting concept classes, including the class of monotone k-term DNF, which are not known to be or are known not to be polynomially learnable (unless $RP = NP$) under arbitrary distributions, are polynomially learnable under a broad class of probability distributions, called simple distributions. This class of distributions includes all enumerable distributions and hence all distributions with bounded precision parameters that can be usually found in statistics books. Sampling is done according to the universal m distribution, or its polynomial-time version, which assign higher probability to 'simpler' examples, where 'simple' means 'of low Kolmogorov complexity'.

Despite their successes, propositional learning approaches suffer from the limited expressiveness of their hypothesis language. Among other issues, propositional languages allow one to ignore the complications arising from recursion and the introduction of new predicates. Furthermore, computational learning theory has mainly focussed on learning in the absence of prior knowledge, whereas difficult learning problems typically require the presence of a substantial body of prior knowledge.

Learning systems that use more expressive languages, typically subsets of first-order logic, have recently attracted a substantial amount of research effort in the machine learning community. As the learned hypothesis most often takes the form of a set of first-order Horn clauses (logic programs), the field has been named Inductive Logic Programming (ILP) [Muggleton 1991, Muggleton 1992]. ILP takes the field of machine learning somewhat closer to a practical method for inducing Turing-equivalent theories.

In this paper we will concentrate on the problem of learning a single concept or *target predicate*. Other work in ILP has focussed on learning several, possibly interdependent, concepts [Shapiro 1983, De Raedt and Bruynooghe 1992]. Few PAC-learnability results have been established for either case (see section 2), although the multiple-

concept learning methods of [Shapiro 1983] and [De Raedt and Bruynooghe 1992] have been shown to identify the correct concept in the limit.

Three distinct but related approaches to ILP have emerged. *Inverse resolution* methods (for example, CIGOL [Muggleton and Buntine 1988] and ITOU [Rouveirol 1991]) are based on the fact that a correct hypothesis must allow a resolution proof of the examples. Inverse resolution is thus a nondeterministic process that generates all possible premises of resolution proofs for the observations. The GOLEM system [Muggleton and Feng 1990] is based on Plotkin's notion of the relative least general generalization (RLGG) of a set of observations with respect to some background knowledge; by making certain syntactic restrictions, it can be shown that a unique, finite RLGG can be found, which is then simplified to generate a reasonable hypothesis. The third class of systems, which includes LINUS [Lavrac et al. 1991] and FOIL [Quinlan 1990], work by extending propositional approaches to a first-order framework. While FOIL uses heuristic search techniques from propositional learning directly, LINUS explicitly converts the first-order representation to a propositional problem by defining appropriate sets of new Boolean features.

We shall use this latter approach to obtain our results, because of the direct applicability of existing results in computational learning theory.[1] Section 2 provides formal definitions for the ILP problem and for the syntactic restrictions we will be considering, and section 3 illustrates these definitions in the context of a simple example. Section 4 shows how an ILP problem can be transformed to a propositional problem, and section 5 gives the principal results. In section 6 we give several examples of determinate logic programs, which imply that many interesting and non-trivial concepts belong to this class. Section 7 discusses ways in which the syntactic restrictions might be relaxed and suggests topics for further research.

2 Inductive Logic Programming

Definition 1 ILP Problem:
Given:

- *A set $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$ of positive $\mathcal{E}^+$ and negative $\mathcal{E}^-$ examples, represented as ground literals.*
- *Background knowledge $\mathcal{B}$, a set of first-order Horn clauses, typically including further facts describing the examples in $\mathcal{E}^+$ and $\mathcal{E}^-$, such that $\mathcal{B} \not\models \mathcal{E}^+$.*

Find:

[1] It should be noted that only positive results can be obtained this way, since the hardness of a transformed propositional problem does not guarantee that the original first-order problem cannot be solved efficiently by some other means.

- *Hypothesis $\mathcal{H}$, also a set of Horn clauses, such that $\mathcal{B} \wedge \mathcal{H} \models \mathcal{E}^+$ (where $\models$ is the relation of logical entailment), and $\mathcal{B} \wedge \mathcal{H} \wedge \mathcal{E}^-$ is consistent.*

We define l to be the number of distinct predicates $q_1 \ldots q_l$ in $\mathcal{B}$, and n to be the arity of the target predicate p.

A brief overview of first-order logic terminology is given in the appendix. Logic programming concepts are extensively covered in the standard textbook by Lloyd [Lloyd 1987]. For reference, a Horn clause (more specifically, a definite program clause [Lloyd 1987]) has the following form:

$$h \leftarrow l_1, l_2, \ldots, l_s$$

where $h, l_1, l_2, \ldots, l_s$ are positive literals. The literal h is called the head and the conjunction of literals $l_1, l_2, \ldots, l_s$ is called the body of the clause. It should be noted that variables in the head of the clause are implicitly universally quantified, while variables that appear in the body, but not in the head are implicitly existentially quantified.

In the logical framework we have adopted a concept is a predicate. When expressed as a logic program, a concept (predicate) definition is a set of clauses each of which has the target predicate appearing in its head:

$$p(\ldots) \leftarrow l_{11}, l_{12}, \ldots.$$
$$p(\ldots) \leftarrow l_{21}, l_{22}, \ldots.$$
$$\ldots$$

Although it appears to be a conjunction of sufficient conditions, predicate completion is assumed in the evaluation of queries, so that the above definition is in fact equivalent to the logical form

$$p(\ldots) \leftrightarrow [l_{11}, l_{12}, \ldots] \vee [l_{21}, l_{22}, \ldots] \vee \ldots$$

Definition 2 k-clause definitions:
A k-clause predicate definition consists of up to k Horn clauses with the same predicate symbol in the head.

As can be seen above, a k-clause predicate definition corresponds to a k-term DNF formula.

In practice, ILP systems work within various other syntactic restrictions in order to limit the complexity of the problem. These are defined as follows:

Definition 3 Ground background knowledge:
$\mathcal{B}$ is ground if it consists of ground unit clauses only.

This restriction is used in both FOIL and GOLEM. Non-ground background knowledge may be used, but has to be converted to a ground model, i.e. a set of ground unit clauses, by carrying out all h-easy derivations starting from the constant symbols in the observations. This has to be done prior to the learning process. In this case, the input size includes the number of unit clauses in the ground version of $\mathcal{B}$. In the LINUS approach, no pre-conversion is done, but instead the values of the propositional features are calculated by queries to $\mathcal{B}$, hence the following assumption:

Definition 4 Efficient background knowledge:
$\mathcal{B}$ is efficient if all atomic queries to it can be answered in time polynomial in the arity of the query predicate.

The syntactic complexity of $\mathcal{B}$ can also be limited:

Definition 5
Bounded-arity background knowledge:
$\mathcal{B}$ is of bounded arity if the maximum arity of the predicates in $\mathcal{B}$ is bounded by some constant j.

We can also define a quantitative measure relating to the complexity of a clause:

Definition 6 Depth of variables:
Consider a clause $h \leftarrow l_1, l_2, ..., l_r, ...$ Variables that appear in the head h have depth zero. Let a variable V appear first in literal l_r. Let d be the maximum depth of any of the other variables in l_r that appear in the clause $h \leftarrow l_1, l_2, ..., l_{r-1}$. Then the depth of V is $d + 1$.

Several types of restrictions can be imposed on the form of the hypothesis itself:

Definition 7 Constrained clauses:
A clause is constrained if all variables in the body also appear in the head.

Definition 8 Function-free clauses:
A clause is function-free if it has no function symbols.

Definition 9 Nonrecursive clauses:
A clause is nonrecursive iff the predicate symbol in its head does not appear in any of the literals in its body. A predicate definition is nonrecursive iff all the clauses in it are nonrecursive.

'Mutually' recursive definitions, such as

$$p(X, Y) \leftarrow r(X, Z), q(Z, Y)$$
$$q(X, Y) \leftarrow s(X, Z), p(Z, Y)$$

are not considered recursive by the above definition, which is, however, sufficient for our purposes, since we are concerned with learning the definition of a single predicate, given the definitions of some other predicates.

Definition 10 Determinate clauses:
A clause is determinate iff each of its literals is determinate; a literal is determinate iff each of its variables

that does not appear in preceding literals has only one possible binding given the bindings of its variables that appear in preceding literals.

Given maximum variable depth i and maximum arity j of predicates from $\mathcal{B}$, the above condition of determinacy is equivalent to the notion of ij-determinacy used in GOLEM [Muggleton and Feng 1990]. A similar idea was later used within FOIL [Quinlan 1991].

Given these definitions, we can state the following prior results. Page and Frisch [Page and Frisch 1992] have shown that a single constrained, nonrecursive clause is learnable. Džeroski and Lavrač [Dzeroski and Lavrac 1992] have shown that a set of constrained, nonrecursive, function-free clauses can be transformed into a polynomially larger propositional representation. The work reported in this paper extends these results, replacing constrained clauses with determinate clauses and allowing recursion.

3 Example

Let us illustrate the above definitions on a simple ILP problem. The task is to define the target predicate $grandmother(X, Y)$, which states that person X is a grandmother of person Y, in terms of predicates from the background knowledge, which includes the predicates $father$ and $mother$. The training examples and background knowledge are given in table 1.

A definition of the target predicate in terms of background knowledge predicates is

$$\forall X \forall Y : grandmother(X, Y) \leftrightarrow$$
$$[\exists Z : father(Z, Y) \wedge mother(X, Z)] \vee$$
$$[\exists U : mother(U, Y) \wedge mother(X, U)]$$

or in logic programming notation

$grandmother(X, Y) \leftarrow$
$\quad father(Z, Y), mother(X, Z).$
$grandmother(X, Y) \leftarrow$
$\quad mother(U, Y), mother(X, U).$

In the above terminology, this hypothesis is *determinate* (but not *constrained*), because each occurrence of a new variable (Z in $father(Z, Y)$ and U in $mother(U, Y)$) has only one possible binding given particular values of the other (old) variables in the literal (Y in this case). The hypothesis is of course function-free, the maximum depth of any variable is 1, and the maximum arity of background knowledge predicates is 2 ($i = 1, j = 2$).

However, the logically equivalent hypothesis

$grandmother(X, Y) \leftarrow$
$\quad mother(X, Z), father(Z, Y).$

$grandmother(X, Y) \leftarrow$
$\quad mother(X, U), mother(U, Y).$

is not determinate, since the new variable Z in the literal $mother(X, Z)$ can have more than one binding for a fixed value of X ($X = ann$, $Z = tom$ or $Z = jim$).

4 Transforming ILP problems to propositional form

Our learnability results are based on transforming an ILP problem to a propositional form, then using learnability results for the propositional case. This can only be done for a restricted class of ILP problems. The features for the propositional version of the problem correspond to all possible applications of the background predicates to the arguments of the target predicate. We first present the transformation algorithm and then illustrate its operation on the example given above.

For the moment, we consider only the problem of learning non-recursive clauses. To transform the ILP problem of constructing a definition for target predicate $p(X_1, X_2, ..., X_n)$ proceed as follows. First, construct a list F of all literals that use predicates from the background knowledge and contain variables of depth at most i. The determinate literals that introduce new variables are excluded from this list, as they do not distinguish between positive and negative examples (the new variables have a unique binding for any example, due to the determinacy restriction). The resulting list is the list of features used for propositional learning. Next, transform the examples to propositional form. For each example, the truth value of each of the propositional features is determined by calls to the background knowledge base. This is done by algorithm 1. Finally, after applying a propositional learning algorithm, transform the output propositional concept definition to Horn clause form. This conversion is fairly straightforward (see below).

Algorithm 1

1. $V_0 = \{X_1, X_2, ..., X_n\}$
2. $L = \{\}$
3. *for* $r = 1$ *to* i *do*
   - D_r = *the set of literals* $q(Y_1, Y_2, ...)$, *where* q *is a predicate in* $\mathcal{B}$ *and the literal contains exactly one new variable not in* V_{r-1},[2] *that are determinate*

[2] The restriction to one new variable is for the purposes of simplicity and does not affect the validity of the general theorems.

Table 1: A simple ILP problem

Training examples	*Background knowledge*		
$grandmother(ann, bob).$	$father(zak, tom).$	$father(pat, ann).$	$father(zak, jim).$
$grandmother(ann, sue).$	$mother(ann, tom).$	$mother(liz, ann).$	$mother(ann, jim).$
$\neg grandmother(bob, sue).$	$father(tom, sue).$	$father(tom, bob).$	$father(jim, dave).$
$\neg grandmother(tom, bob).$	$mother(eve, sue).$	$mother(eve, bob).$	$mother(jean, dave).$

- *add literals from D_r to the end of list L*
- $V_r = V_{r-1} \cup$ $\{Y | Y$ *appears in a literal from* $D_r\}$

4. $F = \{q_s(Y_1, Y_2, ..., Y_{j_s}) |$ $q_s \in \{q_1, q_2, ..., q_l\}, Y_1, Y_2, ..., Y_{j_s} \in V_i\}$ $- (D_1 \cup D_2 ... \cup D_i)$

5. *for each* $p(a_1, a_2, ..., a_n) \in \mathcal{E}^+$ *and each* $\neg p(a_1, a_2, ..., a_n) \in \mathcal{E}^-$ *do*
   - *determine the values of variables in V_i by executing the body of the clause $p(X_1, X_2, ..., X_n) \leftarrow L$ with variables $X_1, ..., X_n$ bound to $a_1, ..., a_n$.*
   - *given the values of the variables in V_i, determine f, the vector of truth values of the literals in F, by querying the background knowledge*
   - *f is an example of a propositional concept c (positive if $p(a_1, a_2, ..., a_n)$ or negative if $\neg p(a_1, a_2, ..., a_n)$)*

As stated earlier, the background knowledge may take the form of a set of ground facts or an oracle (which can be a nonground logic program). In any case, two types of queries have to be posed to the background knowledge. The first type are *existential* queries, which are used to determine the values of the new variables. Given a partially instantiated goal (literal containing variables that do not appear in previous literals), an existential query returns the set (possibly empty) of all bindings for the unbound variables which make the literal true. For example, the query $mother(X, A)$, where $X = ann$ and A is a new variable would return the set of answers $\{A = tom, A = jim\}$. The other type of queries are *membership* queries, where the goal is completely bound. These are used to determine the truth values of the propositional features.

In an actual implementation of algorithm 1 , steps 3 and 5 should be interleaved, i.e. the values of the new variables and the propositional features should be calculated for each example as they are introduced. In this way, the determinacy of literals in step 3 is automatically tested when the existential query determining the values of the new variables in a literal is posed. Should for any of the examples the set of answers to the existential query happen not to be a singleton, the literal under consideration is not determinate.

Finally, it should be noted that information about functional relationships that hold among arguments of background knowledge predicates may be given to the learner to start with. A weaker kind of information, namely information specifying which arguments of a predicate are to be considered as input (old variables) and which as output (either old or new variables) is used in both FOIL [Quinlan 1990] and GOLEM [Muggleton and Feng 1990]. The arguments of the background knowledge predicates may also be sorted, as in LINUS [Lavrac et al. 1991], in which case the information about the sorts of variables (arguments) greatly reduces the number of propositional features involved.

Example: For our simple ILP example, we have $j = 2$, $i = 1$, $l = 2$, and $n = 2$. A literal $father(X, A)$, where X is old and A is new is not determinate, as a man can have several children. However, if instead A is old and X is new, the literal is determinate, since each person has exactly one father. As the target predicate is $grandmother(X, Y)$, we have $V_0 = \{X, Y\}$ and $D_1 = \{f(U, X), f(V, Y), m(W, X), m(Z, Y)\}$, where f and m stand for $father$ and $mother$, respectively.

This gives $L_1 = D_1$, $V_1 = \{X, Y, U, V, W, Z\}$. F includes literals such as $f(X, X), f(X, Y), f(Z, Y), f(W, X)$ and similarly $m(Z, Z), m(V, Y), m(W, W), m(U, X)$. In fact, the pairs of arguments of f and m are all the pairs from the Cartesian product $V_1 \times V_1$, excluding the pairs that produce literals from D_1.

To illustrate the transformation process, table 2 gives two features and their propositional values, as well as the values of the variables introduced by the determinate literals, generated for the ILP problem as defined by the training examples and background knowledge from table 1.

To outline the transformation from a propositional DNF concept description to a Horn clause predicate definition we again use our simple example. Suppose a propositional learner induces the concept $c \leftrightarrow x_1 \vee x_2$ from the examples in table 2. The literals $m(X, V)$ and $m(X, Z)$ correspond to features x_1 and x_2. As these literals use new variables, we must include the determinate literals that introduced the new variables in the correspond-

Table 2: Propositional form of a simple ILP problem

			Propositional features							
$g(X,Y)$	X	Y	$f(U,X)$	$f(V,Y)$	$m(W,X)$	$m(Z,Y)$	...	$m(X,V)$	$m(X,Z)$	...
c			U	V	W	Z		x_1	x_2	
1	*ann*	*bob*	*pat*	*tom*	*liz*	*eve*		1	0	
1	*ann*	*sue*	*pat*	*tom*	*liz*	*eve*		1	0	
0	*bob*	*sue*	*tom*	*tom*	*eve*	*eve*		0	0	
0	*tom*	*bob*	*zak*	*tom*	*ann*	*eve*		0	0	

ing first-order form. The corresponding first-order form would then be

$$\forall X \forall Y : grandmother(X,Y) \leftrightarrow$$
$$[\exists V : f(V,Y) \wedge m(X,V)] \vee$$
$$[\exists Z : m(Z,Y) \wedge m(X,Z)]$$

5 Results

The following result is derived from similar results by Džeroski and Lavrač [Dzeroski and Lavrac 1992] and Muggleton and Feng [Muggleton and Feng 1990].

Theorem 1 *Algorithm 1 transforms the ILP problem defined by a set of m examples $\mathcal{E}$ of the target predicate $p(X_1, X_2, ..., X_n)$, background predicates $q_1, q_2, ..., q_l$ of maximum arity j, and maximum depth of variables i, to a propositional form in time*

$$\mathcal{O}(poly(j)ml((jl+1)n)^{j^{i+1}}),$$

assuming that each call to a predicate from the background knowledge takes $\mathcal{O}(poly(j))$ to answer.

Proof: Let $v_r = |V_r|$. We have $v_0 = n$ and $v_r \leq v_{r-1} + jlv_{r-1}^{j-1}$, since there are at most jv_{r-1}^{j-1} applications of each predicate from the background knowledge, each of them introducing exactly one new variable. It can be easily proved that $v_i \leq ((jl+1)n)^{j^i}$. Each of the l predicates from the background knowledge can be applied to the variables in V_i in at most v_i^j ways. The number of features in F is thus upper bounded by $n_i = lv_i{}^j \leq l((jl+1)n)^{j^{i+1}}$.

The transformation of a single example to propositional form takes $\mathcal{O}(poly(j)v_i)$ time to determine the values of variables in V_i and $\mathcal{O}(poly(j)n_i)$ time to determine the truth values of features in F, altogether $\mathcal{O}(poly(j)l((jl+1)n)^{j^{i+1}})$. For m examples, the transformation process takes $\mathcal{O}(poly(j)ml((jl+1)n)^{j^{i+1}})$ time. □

Theorem 2 *k-clause predicate definitions consisting of non-recursive determinate function-free Horn clauses with variables of bounded depth are polynomially PAC-learnable under simple distributions.*

Proof: After transforming the problem to a propositional form, we use the algorithm for learning monotone k-term-DNF described in [Li and Vitanyi 1991]. When transforming the induced monotone k-term-DNF formula back to first-order form, we have to include the necessary determinate literals which determine the variables used in the features from the induced formula. This may change the length of each of the clauses, but not the number of clauses. The time this transformation takes is $\mathcal{O}(hjn_i)$, where h is the size of the induced monotone k-term-DNF formula. The learnability under simple distributions of k-clause predicate definitions consisting of non-recursive determinate function-free Horn clauses with variables of depth up to i then follows from the polynomial learnability of monotone k-term-DNF under simple distributions. □

Theorem 3 *k-clause (possibly recursive) predicate definitions consisting of determinate function-free Horn clauses with variables of bounded depth are polynomially PAC-learnable under simple distributions if we restrict the arity of the target predicate $p(X_1, X_2, ..., X_n)$ to be less than j and allow $\mathcal{O}(m((jl+j+1)j)^{j^{i+1}})$ existential and membership queries about the target predicate.*

Proof: Since we have restricted the arity of the target predicate to be less than j, we can use the target predicate as any of the other predicates from the background knowledge, except that the literal (feature) $p(X_1, X_2, ..., X_n)$ is deleted from F. Thus the number of predicates in the background knowledge has increased to $l' = l + 1$. During the transformation process, when generating values for features involving p for the training examples, the training examples are used first. For example, suppose the value of a feature is to be assigned the truth-value of the fact $p(a_1, a_2, ..., a_n)$. If this fact (or its negation $\neg p(a_1, a_2, ..., a_n)$) is among the training examples we assign the appropriate truth value to the feature. However, as we do not have an exhaustive set of training examples for p, it may happen that the fact $p(a_1, a_2, ..., a_n)$ is not among the training examples. In that case we have to resort to membership queries in order to complete the transformation process. In addition, if we allow for recursive literals that introduce new variables, existential membership queries will have

to be used, which would determine the values for the new variables. Existential queries have also been used in MIS [Shapiro 1983].

Following an argument similar to the one in the proof of theorem 1, we see that the number of features involving p is of the order of $n_i^j \leq [((jl'+1)n)^{j^i}]^n \leq ((jl+j+1)j)^{j^{i+1}}$. In the worst case, we will have to make a membership query for each of the features for each of the training examples, a total of $\mathcal{O}(m((jl+j+1)j)^{j^{i+1}})$ membership queries. A similar result is obtained for the determinate literals involving p and existential queries. Thus, we have transformed the ILP problem into a propositional form and the learnability under simple distributions of k-clause, possibly recursive, predicate definitions consisting of determinate function-free Horn clauses with variables of depth up to i then follows from the polynomial learnability of monotone k-term-DNF under simple distributions. □

It should be noted that the induced recursive definitions may be highly inefficient or even non-terminating if executed by a PROLOG interpreter. Additional information and techniques, similar to the ones used in FOIL, may be used to prevent the construction of such definitions.

6 On the expressiveness of determinate logic programs

Despite the number of syntactic restrictions that were imposed on predicate definitions in the previous sections, the class of determinate logic programs includes many interesting and nontrivial concept definitions. We support this claim by listing several determinate predicate definitions that were actually learned by the ILP systems GOLEM and FOIL. These include concepts from integer arithmetic and list manipulation.

$less\_than(A, B) \leftarrow$
    $successor(A, B)$.
$less\_than(A, B) \leftarrow$
    $successor(A, C), less\_than(C, B)$.

The above definition is a 2-clause recursive predicate definition, where the maximum arity is $j = 2$ and the maximum variable depth is $i = 1$. It was learned from training examples of the target predicate and background knowledge about the predicate *successor*.

The following definition has the same complexity parameters as above, except for the maximum arity, which is in this case $j = 3$. The background knowledge consists of the predicates *plus* and *decrement*. In addition, we need the unary predicate *zero*, since no function symbols (and thus constants) are allowed in the learned clauses.

$multiply(A, B, C) \leftarrow$
    $zero(B), zero(C)$.
$multiply(A, B, C) \leftarrow$
    $decrement(B, D)$,
    $plus(A, E, C)$,
    $multiply(A, D, E)$.

Finally, consider the recursive formula for computing the number of combinations consisting of m elements chosen out of n elements.

$$\binom{n}{0} = 1$$

$$\binom{n}{m} = \frac{n}{m}\binom{n-1}{m-1}$$

It is implemented by the following two clauses:

$choose(A, B, C) \leftarrow$
    $zero(B), one(C)$.
$choose(A, B, C) \leftarrow$
    $decrement(B, D)$,
    $decrement(A, E)$,
    $multiply(B, C, G)$,
    $divide(G, A, F)$,
    $choose(E, D, F)$.

The maximum variable depth is in this case $i = 2$, the maximum arity is $j = 3$, and $k = 2$.

The notion of determinacy should not be confused with the notion of determinism. Consider, for example the following logic program.

$member(A, B) \leftarrow$
    $components(B, C, D)$,
    $= (A, C)$.
$member(A, B) \leftarrow$
    $components(B, C, D)$,
    $member(A, D)$.

Since the clauses are to contain no function symbols, a flattened version of the '.' function symbol is provided as a background knowledge predicate ($components(A, B, [A|B])$), as well as the predicate $null(X)$, which is true for the empty list only ($null([])$). (In PROLOG $[A|B]$ stands for $.(A, B)$, and [] stands for the empty list.) In addition, the equality predicate $=$ is provided ($X = X$).

Given the above definition for the *member* predicate, the query $member(X, [1, 2, 3])$ has more than one cor-

rect answer (three in this case: $X = 1, X = 2, X = 3$). The definition is thus nondeterministic. At the same time, it is determinate, with $j = 3, i = 1$.

We conclude with the *quicksort* example, where the task is to learn how to sort a list, given the background knowledge predicates $partition(A, B, C, D)$ and $append(A, B, C)$. The former takes a number A and a list B, and partitions the list B in two lists C and D, such that all the elements of B which are less than A go to C, and all the others to D. The latter takes two lists A and B and concatenates them to produce C. Using in addition the background knowledge from the *member* definition, the definition of *quicksort* may be expressed as follows [Quinlan 1991]:

$quicksort(A, B) \leftarrow$
$\quad = (A, B), null(A).$
$quicksort(A, B) \leftarrow$
$\quad components(A, C, D),$
$\quad partition(C, D, E, F),$
$\quad quicksort(E, G), quicksort(F, H),$
$\quad append(G, I, B),$
$\quad components(I, C, H).$

The maximum variable depth is in this case $i = 4$ and the maximum background predicate arity is $j = 4$.

To summarize, the above examples illustrate the fact that the class of determinate logic programs includes many interesting and nontrivial concept definitions. We would again like to emphasize that the above definitions have been actually generated by existing ILP systems. It should also be noted, however, that the sets of training examples for the above cases given to FOIL and GOLEM have been exhaustive (for example, for *quicksort* all lists of elements 0..3 of length up to 3 appear as training examples [Quinlan 1991]). In this way, the need for either existential or membership queries about the target predicate has been avoided. This would not be the case if the training examples are randomly chosen.

7 Further work

Determinacy is a reasonable restriction if we are working in a programming domain (as in the above examples); however, in cases such as the induction of scientific theories or mathematical conjectures, indeterminate clauses are common. If we consider a slight variant on the grandmother theory, the complexity problem begins to become apparent:

$greatgrandmother(X, Y) \leftarrow$
$\quad parent(Z_1, Y), parent(Z_2, Z_1), mother(X, Z_2).$

With two possible values for Z_1 and two for Z_2, we will get four times as many propositional features. With additional parameters limiting the length of these indeterminate chains and the number of possible instantiations at each step, we could allow for limited indeterminacy without sacrificing too much efficiency.

Removal of the function-free restriction is straightforward, because any clause containing function symbols can be *flattened*, that is, rewritten in determinate function-free form with the addition of one background clause per function symbol [Rouveirol 1991].

In the above, we have not addressed the question of using new predicates (that is, predicates not appearing in the background knowledge) in the hypothesis. At present it is not known whether new predicates are essential in order to find compact representations for some data sets. Some propositional methods that generate new predicates (e.g., DUCE [Muggleton 1987]) could perhaps be used on the transformed problem, although we have not investigated this possibility.

As mentioned above, the transformational approach does not allow one to use existing propositional hardness results for ILP problems. Page and Frisch used the RLGG approach to show positive results for constrained clauses, also showing negative results for sorted theories. All positive results so far obtained have, nonetheless, been for cases in which the propositional version of the problem is only polynomially larger than the first-order form. A result showing learnability of a class of first-order formulæ that provides exponential compression would be a more conclusive validation of the ILP approach.

Acknowledgements

Sašo Džeroski is on leave from Institut Jožef Stefan, Ljubljana, Slovenia and is supported by a scholarship from the British Council. Stephen Muggleton is supported by a Postdoctoral Fellowship from the UK Science and Engineering Research Council. Stuart Russell is on sabbatical from the Computer Science Division, University of California, Berkeley and is supported by a Visiting Fellowship from the UK Science and Engineering Research Council and an NSF Presidential Young Investigator Award.

References

[Benedek and Itai 1988] G. Benedek and A. Itai. Learnability by fixed distributions. In *Proc. First ACM Workshop on Computational Learning Theory*, pages 80–90, Morgan Kaufmann, San Mateo, CA, 1988.

[De Raedt and Bruynooghe 1992] L. De Raedt and M. Bruynooghe. Interactive concept learning and con-

structive induction by analogy. *Machine Learning*, 8(2): 107–150, 1992.

[Dzeroski and Lavrac 1992] S. Džeroski and N. Lavrač. Refinement graphs for FOIL and LINUS. In S. H. Muggleton, editor, *Inductive Logic Programming*, Academic Press, London, 1992. In press.

[Haussler 1988] D. Haussler. Quantifying inductive bias: AI learning algorithms and Valiant's model. *Artificial Intelligence*, 36(2): 177-221, 1988.

[Lavrac et al. 1991] N. Lavrač, S. Džeroski and M. Grobelnik. Learning nonrecursive definitions of relations with LINUS. In *Proc. Fifth European Working Session on Learning*, pages 265–281, Springer, Berlin, 1991.

[Lloyd 1987] J. W. Lloyd. *Foundations of Logic Programming* (2nd edn), Springer, Berlin, 1987.

[Li and Vitanyi 1991] M. Li and P. Vitányi. Learning simple concepts under simple distributions. *SIAM Journal of Computing*, 20(5): 911–935, 1991.

[Muggleton 1987] S. H. Muggleton. Duce, an oracle-based approach to constructive induction. In *Proc. Tenth International Joint Conference on Artificial Intelligence*, pages 287–292, Morgan Kaufmann, San Mateo, CA, 1989.

[Muggleton 1991] S. H. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4): 295–318, 1991.

[Muggleton 1992] S. H. Muggleton, editor. *Inductive Logic Programming*, Academic Press, London, 1992. In press.

[Muggleton and Buntine 1988] S. H. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proc. Fifth International Conference on Machine Learning*, pages 339–352, Morgan Kaufmann, San Mateo, CA, 1988.

[Muggleton and Feng 1990] S. H. Muggleton and C. Feng. Efficient induction of logic programs. In *Proc. First Conference on Algorithmic Learning Theory*, pages 368–381, Ohmsha, Tokyo, 1990.

[Page and Frisch 1992] C. D. Page and A. M. Frisch. Generalization and learnability: a study of constrained atoms. In S. H. Muggleton, editor, *Inductive Logic Programming*, Academic Press, London, 1992. In press.

[Quinlan 1990] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3): 239–266, 1990.

[Quinlan 1991] J. R. Quinlan. Knowledge acquisition from structured data - using determinate literals to assist search. *IEEE Expert* 6(6): 32–37, 1991.

[Rivest 1987] R. Rivest. Learning decision lists. *Machine Learning*, 2(3): 229–246, 1987.

[Rouveirol 1991] C. Rouveirol. Completeness for inductive procedures. In *Proc. Eighth International Workshop on Machine Learning*, pages 452–456, Morgan Kaufmann, San Mateo, CA, 1991.

[Shapiro 1983] E. Y. Shapiro. *Algorithmic Program Debugging*, The MIT Press, Cambridge, MA, 1983.

[Valiant 1984] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11): 1134–1142, 1984.

Appendix

Formulæ in first order predicate calculus

A variable is represented by an upper case letter followed by a string of lower case letters and digits. A function symbol is a lower case letter followed by a string of lower case letters and digits. A predicate symbol is a lower case letter followed by a string of lower case letter and digits. The negation of F is $\neg F$ or $\overline{F}$.

A variable is a term, and a function symbol immediately followed by a bracketed n-tuple of terms is a term. Thus $f(g(X), h)$ is a term when f, g and h are function symbols and X is a variable. A predicate symbol immediately followed by a bracketed n-tuple of terms is called an atomic formula, or atom. Both l and $\overline{l}$ are literals whenever l is an atomic formula. In this case l is called a positive literal and $\overline{l}$ is called a negative literal.

A clause is a formula of the form

$$\forall X_1 \forall X_2 ... \forall X_s (l_1 \vee l_2 \vee ... l_m)$$

where each l_i is literal and $X_1, X_2, ..., X_s$ are all the variables occurring in $l_1 \vee l_2 \vee ... l_m$. A clause can also be represented as a finite set (possibly empty) of literals. Thus the clause $(l_1 \vee l_2 \vee .. \overline{l_i} \vee \overline{l_{i+1}} \vee ...)$ is equivalently represented as $\{l_1, l_2, .. \overline{l_i}, \overline{l_{i+1}}, ...\}$ or most commonly as $l_1, l_2, .. \leftarrow l_i, l_{i+1}, ...$ A Horn clause is a clause which contains at most one positive literal. A definite program clause is a clause which contains exactly one positive literal. The positive literal in a definite program clause is called the head of the clause while the negative literals are collectively called the body of the clause. A Horn clause with no positive literal is a definite goal. A unit clause is a clause of the form $l \leftarrow$, that is, a definite program clause with an empty body.

A set of clauses is called a clausal theory. A clausal theory represents the conjunction of its clauses. Thus the clausal theory $\{C_1, C_2, ...\}$ can be equivalently represented as $(C_1 \wedge C_2 \wedge ...)$. A set of definite program clauses is called a definite logic program.

Literals, clauses and clausal theories are all well-formed-formulæ (wff's). Let E be a wff or term. $vars(E)$ denotes the set of variables in E. E is said to be ground if and only if vars$(E) = \emptyset$.