Functional Inductive Logic Programming with Queries to the User

F. Bergadano[1] and D. Gunetti[2]

[1]University of Catania, via A. Doria 6/A,
95100 Catania, Italy, bergadan@mathct.cineca.it

[2]University of Torino, corso Svizzera 185,
10149 Torino, Italy, gunetti@di.unito.it

Abstract

The FILP learning system induces functional logic programs from positive examples. For every predicate P, the user is asked to provide a mode (input or output) for each of its argument, and the system assumes that the mode corresponds to a total function, i.e., for a given input there is one and only one corresponding output that makes the predicate true. Functionality serves two goals: it restricts the hypothesis space and it allows the system to ask existential queries to the user. By means of these queries, missing examples can be added to the ones given initially, and this makes the learned programs complete and consistent and the system adequate for learning multiple predicates and recursive clauses in a reliable manner.

1 Introduction

Recently there has been a growing interest, within the Machine Learning community, in the problem of learning logic programs from positive and negative examples in the form of ground literals. The obtained results should naturally be communicated and proposed as important tools for Logic Programming and even for Software Engineering at large. However, this has not yet happened. The reason is, we think, twofold.

First, learning logic programs is difficult, and systems tend to be slow and do not always terminate successfully, even when a solution program exists. A common way to handle this problem consists in restricting the hypothesis space by means of strong constraints of various kinds. In this paper we follow the same idea, and restrict the inductive hypotheses to logic programs that are *functional*, i.e. such that each n-ary predicate can be associated to a total function as follows: m of its arguments are labeled as input, while the remaining n-m are labeled as output, and for every given sequence of input values, there

Acknowledgement: This work was partially supported by ESPRIT project BRA 6020 on Inductive Logic Programming.

is one and only one sequence of output values that makes the predicate true. Functionality constraints have been used before [9, 7, 6, 3]; in the present paper we employ them to query the user for missing examples and explicitly address the problem of consistency and completeness.

Second, the kinds of programs that are learned are usually very simple and often limited to clauses defining just one predicate. Few systems [8, 2, 5] are able to learn programs for multiple predicates, while even beginning Prolog programmers write programs with different clause consequents. This is due, in part, to the need of learning clauses one at a time and independently of each other. If we want to learn a program for predicates P and Q, and we try to construct a clause antecedent for P where Q occurs, then Q must have been defined by the user, or determined extensionally, by means of all of its relevant examples. Something similar occurs with recursion, i.e. for the case when Q=P. We will show in this paper that, as a consequence of the extensional interpretation of recursion and sub-predicates, systems may be unable to learn a program, even when an allowed inductive hypothesis that is consistent with the examples exists. Even worse, it may happen that a program is learned that computes wrong outputs even for the *given* examples.

The FILP system, presented here, solves this problem by querying the user for any example that may be missing, depending on the hypothesis space that has been defined. The queries that are asked to the user are of the type of the existential queries of CLINT [5] and MIS [8], because they contain unbound variables. However, in FILP learning is one-step and example completion is done in a preprocessing phase.

2 The FILP System

Since FILP learns functional relations, it really only needs positive examples. Negative examples are implicitly assumed to be all the ones having the same input values as the positive examples but different output values. In the sequel, by *example* we usually mean *positive example*, while α and γ represent generic conjunctions of literals.

It is well known that in logic programming variables have not a fixed role: they can act as input or output variables as desired. For example, the predicate *append*(X,Y,Z) can be used with mode *append*(in,in,out) to append two lists, or with mode *append*(out,out,in) to split a list in two sublists.
On the other hand, if we want to learn functional logic programs (logic programs whose input-output behavior is functional) we need to specify a (functional) mode for every variable of every literal used in the learning task, in order to employ and learn only functional relations. For example *append*(in,in,out) would be a legal way to use append, but *append*(out,out,in) would not, because it does not represent a function. On this ground, in our system we ask the user to provide a functional mode for all predicates, and then we use it for constraining the allowed clauses as follows:
1) Suppose Q and P have mode Q(in,out) and P(in,out); the literal Q(W,Z) can

occur in an intermediate clause P(X,Y) :- α, Q(W,Z), γ iff either (a) W=X (i.e., the input is bound because it is passed as input in the head of the clause) or (b) W occurs in α (i.e., it is computed before Q is called) [10];
2) A clause is in an acceptable final form only if the output variables of its head occur in the body, i.e., only if the output is not left unbound.

Moreover, all clauses are required to be function-free. This can be achieved by means of a flattening procedure [7]. A basic version of FILP without queries (BFILP) follows the algorithmic scheme of FOIL [4]:

Basic FILP:
For all inductive predicates P do
while examples(P) $\neq \emptyset$ do
  Generate one clause "P($\vec{X}$) :- α"
  examples(P) $\leftarrow$ examples(P) $-$ covered(α)

Generate one clause:
$\alpha \leftarrow$ true
while covered(α) $\neq \emptyset$ do
  if consistent(α) then return(P($\vec{X}$) :- α)
  else choose a predicate Q and its arguments Args
    such that the functionality constraint is satisfied
    if no such Q is found then backtrack
    $\alpha \leftarrow \alpha \wedge$ Q(Args)

Where every predicate Q can be defined by the user (intensionally) by means of logical rules or (extensionally) simply giving some examples of its input-output behavior. In particular, clauses can be recursive and, in this case, Q = P, and its truth value can only be determined by the available examples.

Definition 1: We say that the clause P(X,Y) :- α(X,Y) *extensionally covers* P(a,b) iff α(a,Y) *extensionally computes* Y = b, where extensional computation is defined as follows:

- α = Q(a,Y) with functional mode Q(in,out). Then Q(a,Y) *extensionally computes* Y = b iff Q(a,b) is derivable from the definition of Q or is a given example of Q.
- $\alpha = \gamma$(X,T), Q(T,Y) with functional mode γ(in,out) and Q(in,out). Then γ(a,T), Q(T,Y) *extensionally computes* Y = b iff γ(a,T) *extensionally computes* T = e and Q(e,b) is derivable from the definition of Q or is a given example of Q.

In the algorithm, an example P(a,b) belongs to covered(α) iff α(a,Y) extensionally computes Y=b, and consistent(α) is true iff, for no such example, α(a,Y) extensionally computes Y=c and c$\neq$b. The choice of the literal Q(Args) to be added to the partial antecedent α of the clause being generated is guided by heuristic information. It might nevertheless be a wrong choice in some cases, in the sense that it causes the procedure "Generate one clause" to fail by exiting the while loop without returning any clause. This problem is remedied by making the choice of Q(Args) a backtracking point.

In the worst case, all possible literals will be tried every time, and the complexity is exponential in the number of these literals. We view this problem as intrinsic of induction and unavoidable - the only thing we can do is reduce the number of possible clauses by means of strong constraints given a priori by the user. An advantage of extensional methods is that clauses are generated independently of each other. As a consequence we must search the space of possible clauses (exponential in the number of possible literals), not the space of possible logic programs (= *sets* of possible clauses). This independence of the clauses is made possible by the extensional interpretation of recursion and of the other inductive relations: when a predicate Q corresponding to an inductive relation occurs in a clause antecedent α, it is evaluated as true when the arguments match one of the positive examples. The method leads to a fundamental property of extensional methods (proofs are found in [1]).

Definition 2: A program P is *complete* w.r.t. the examples E iff ($\forall$ Q(i,o) $\in$ E) P $\vdash$ Q(i,o). A program P is *consistent* w.r.t. the examples E iff ($\nexists$ Q(i,o) $\in$ E) P $\vdash$ Q(i,o') and o$\neq$o'.

Lemma 1: Suppose BFILP successfully exits its main loop and outputs a logic program P, that always terminates (w.r.t. SLD-resolution) for the given examples. Let Q(X,Y) :- α(X,Y) be any clause of P, then
($\forall$Q(a,b)$\in$Examples(Q)) α(a,Y) ext. computes Y=b $\rightarrow$ P $\vdash$ Q(a,b).

Theorem 1: If BFILP terminates successfully, then it outputs a complete program P.

The above proof is also valid for systems such as FOIL, and is a partial justification of the extensional evaluation of the generated clauses. However, extensionality forces us to include many examples, which would otherwise be unnecessary. In fact other desirable properties, similar to the one given by Theorem 1, are not true:

1) For a complete and consistent logic program P, it may happen that P $\vdash$ Q(a,b), but none of its clauses extensionally cover Q(a,b). As a consequence BFILP would be unable to generate P, and would not terminate successfully. Consider this program P:

reverse(X,Y) :- *null*(Y), *null*(X).
reverse(X,Y) :- *head*(X,H), *tail*(X,T), *reverse*(T,W), *append*(W,[H],Y).

Let *reverse*([a,b],[b,a]) be the only given example. This example follows from P (P $\vdash$ *reverse*([a,b],[b,a])) but is not extensionally covered: the first clause does not cover it because *null*([a,b]) is false, and the second clause does not cover it extensionally because *head*([a,b],a) and *tail*([a,b],[b]) are true, but *reverse*([b],[b]) is not in examples(*reverse*).

2) Let P be a program to compute a function Q and Q(i,o) ∈ examples(Q). Even if, for all clauses Q :- α in P, consistent(α) is true, it may still happen that P ⊢ Q(i,o') with o ≠ o'. In other words BFILP might generate a program that is not consistent even for the given examples. Consider the following program P:

reverse(X,Y) :- *head*(X,H), *tail*(X,T1), *head*(Y,H), *tail*(Y,T2), *reverse*(T1,T2).
reverse(X,Y) :- *null*(X), *null*(Y).
reverse([X,Y,Z],[Z,Y,X]).

which can be learned by BFILP with this set of examples:
reverse([],[]), *reverse*([1],[1]), *reverse*([3,2,1],[1,2,3]).
Then P ⊢ *reverse*([3,2,1],[3,2,1]). Nevertheless, *reverse*([3,2,1],[3,2,1]) is not extensionally covered by the first clause. In fact, *reverse*([2,1],[2,1]) is not given as an example. In order to prevent BFILP from generating that inconsistent program, in this case we must tell the system that *reverse*([2,1],[2,1]) is wrong. This is done by adding a positive example, namely *reverse*([2,1],[1,2]).

To overcome the above problems, FILP queries the user for some of the missing examples. Every legal clause (= permitted by the constraints) of the type "P(X,Y) :- A(X,W), Q(X,W,Z), α." where Q is an inductive predicate with mode Q(in,in,out), is processed with the following procedure:

for every example P(a,b) do
  extensionally compute A(a,W), obtaining a value W = c
  ask the user for the value Z computed by Q(a,c,Z)
  add this example to examples(Q)

This procedure must be repeated for every clause, again and again, until no more examples are added for the inductive predicates. Both for making the above procedure terminate and for guaranteeing the termination of learned programs, we require that any recursive call within a generated clause matches the following pattern: "P(X_1, ..., X_i, ..., X_n) :- ..., Q(X_i,Y), ..., P(X_1, ..., Y, ..., X_n), ..." where Q(X,Y) is known to define a well ordering between Y and X (Y<X). A similar technique is found in [4], but does not guarantee termination on new examples. It is possible to show that, if every recursive clause in P satisfies the above constraint, then the example completion procedure terminates.

As an instance, suppose that we want to learn *reverse*. Consider the clause *reverse*(X,Y) :- *tail*(X,T), *reverse*(T,W). It satisfies the constraint on recursive calls because, when *tail*(X,T) is true, then T is a shorter list than X and this is a well order relation. Consider the example *reverse*([a,b,c],[c,b,a]). By using the clause, the user is queried for the value of *reverse*([b,c],W), and this is added to examples(*reverse*). This new example causes the repetition of the procedure, and the user is queried for *reverse*([c],W), and at the next step for *reverse*([],W).

Lemma 2: Suppose the examples given to an extensional learning system are completed with the above completion procedure. Suppose also that some pro-

gram P belongs to the hypothesis space and Q(a,b)∈examples(Q) after the completion.
If P ⊢ Q(a,b) then the first clause in P resolved against Q(a,b) extensionally covers Q(a,b).

Theorem 2: If a complete and consistent program P exists, then FILP will terminate successfully.

Theorem 3: If FILP terminates successfully, then it outputs a consistent program P.

By virtue of Theorem 1, this program will also be complete.

References

[1] F. Bergadano and D. Gunetti. Sufficient and Correct Induction of Functional Logic Programs. *Tech. Rep. 92.9.2, CS Dept., Univ. of Torino*, 1992.

[2] J. U. Kietz and S. Wrobel. Controlling the Complexity of Learning in Logic through Syntactic and Task-Oriented Models. In *Proc. Workshop on Inductive Logic Programming*, pages 107–126, 1991.

[3] N. Lavrac, S. Dzeroski, and M. Grobelnik. Learning nonrecursive definitions of relations with linus. In Y. Kodratoff, editor, *Proc. of the Machine Learning-EWSL 91*, pages 265–281, Porto, Portugal, 1991. Springer-Verlag.

[4] R. Quinlan. Knowledge Acquisition from Structured Data. *IEEE Expert*, 6(6):32–37, 1991.

[5] L. De Raedt and M. Bruynooghe. CLINT: a Multistrategy Interactive Concept-Learner and Theory Revision System. In *Proc. Workshop on Multistrategy Learning*, pages 175–190, 1991.

[6] L. De Raedt and Maurice Bruynooghe. Belief Updating from Integrity Constraints and Queries. *Artificial Intelligence*, 53:291–307, 1992.

[7] C. Rouveirol. Flattening: a Representation Change for Generalization. *Machine Learning*, 1993. Special issue on Evaluating and Changing Representation, K. Morik, F. Bergadano and W. Buntine (Eds.).

[8] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[9] I. Stahl, B. Tausend, and R. Wirth. General-to-specific learning of horn clauses from positive examples. In P. Dewilde and J. Vanderwalle, editors, *Proc. of the CompEuro, 1992*, pages 436–441, The Hague, Netherlands, 1992. IEEE Comp. Soc. Press.

[10] R. Wirth and P. O'Rorke. Constraints on predicate invention. In L. A. Birnbaum and G. C. Collins, editors, *Proc. of the 8th Int. Workshop on ML*, pages 457–461, Evanston, Illinois, 1991. Morgan Kaufmann.