Learning Multiple Predicates

Antonis Kakas[1], Evelina Lamma[2], Fabrizio Riguzzi[2]

[1] Department of Computer Science, University of Cyprus
75 Kallipoleos str., CY-1678 Nicosia, Cyprus
antonis@turing.cs.ucy.ac.cy
[2] DEIS, Università di Bologna,
Viale Risorgimento 2, I-40136 Bologna, Italy,
{elamma,friguzzi}@deis.unibo.it

Abstract. We present an approach for solving some of the problems of top-down Inductive Logic Programming systems when learning multiple predicates. The approach is based on an algorithm for learning abductive logic programs. Abduction is used to generate additional information that is useful for solving the problem of global inconsistency when learning multiple predicates.

1 Introduction

Most logic programs contain the definition of several predicates. However, most Inductive Logic Programming (ILP) systems have been designed for learning definitions for a single predicate and they find problems when they are employed for learning multiple predicates.

The simplest approach for learning multiple predicates consists in iteratively performing a single predicate learning task. However, this approach suffers from various problems [5]: it is sensitive to the order in which predicate are learned, an overgeneral definitions for a predicate p can prevent the system from finding a definition for another predicate q that depend on p and it is not possible to learn mutually recursive predicates.

In order to learn mutually recursive predicates, the learning of clauses for different predicates must be interleaved. In this case, however, a top-down covering algorithm faces the problem that adding a consistent clause to a partial hypothesis can make previous clauses for other predicates inconsistent [5]. Therefore, expensive re-testing of examples and backtracking on clause addition to the theory must be performed.

In order to overcome these problems, many top-down systems (e.g., ICN [16], MULT_ICN [15], FOIL [19], FOCL [17], MIS [20] with the lazy strategy) use *extensional coverage* instead of *intensional coverage*. In intensional coverage, a clause is evaluated by performing a derivation of each example from a program composed by the clause, the background knowledge and the clauses previously learned. In extensional coverage, the atoms in the training set are used as a definition for the target predicates, instead of the clauses previously learned. In this way, clauses are learned independently from each other. We will distinguish between *extensional* and *intensional systems* depending on the type of coverage they use. However, extensional coverage introduces other problems because the learning algorithm can be unsound: the learned theory can be both inconsistent and incomplete, as shown in [5].

In this paper, we propose the algorithm MPL-A (Multiple Predicate Learning by Abduction) that is able to learn definite clause programs containing the definition of multiple predicates by interleaving the learning of clauses for different predicates. The algorithm exploits abduction in order to overcome the problems of intensional systems while partially avoiding the pitfalls of extensional systems.

The algorithm we propose is obtained by modifying those presented in [11, 12] for learning abductive logic programs. The problem of learning abductive logic programs has recently received some attention. A number of works [14, 11, 12] have started to appear on the subject, and, more generally, on the relation existing between abduction and induction and how they can integrate and complement each other [6, 7, 1].

In order to cope with global inconsistency when learning multiple predicates, in this work we exploit abduction for testing the coverage of negative examples by generating negative assumptions about target predicates whose definition is currently incomplete. These assumptions ensure that the tested negative examples will not be covered. They are then added to the training set and clauses learned afterwards are tested against them. In this way the addition of a new clause will not make previous clauses inconsistent.

The paper is organized as follows: in section 2 we recall the basics of Inductive Logic Programming and we discuss the problems of ILP systems when learning multiple predicates. Section 3 presents the concepts of Abductive Logic Programming (ALP) that are needed for the algorithm. Section 4 presents the learning algorithm and section 5 shows some experiments performed with the system. In section 6 we discuss related work and in section 7 we conclude and present the directions for future work.

2 Inductive Logic Programming

Now we recall some basics on ILP. We first give a definition of the ILP problem [2]:

Given:

a set $\mathcal{P}$ of possible programs
a set E^+ of positive examples
a set E^- of negative examples
a consistent logic program B such that
$B \not\vdash e^+$ for at least one $e^+ \in E^+$.

Find:

a logic program $P \in \mathcal{P}$ such that
$\forall e^+ \in E^+$, $B \cup P \vdash e^+$ (*completeness*)
$\forall e^- \in E^-$, $B \cup P \not\vdash e^-$ (*consistency*).

With a great deal of approximation, top-down ILP systems share a common basic algorithm [2]:

$T := \emptyset$
while $E^+ \neq \emptyset$ do (Covering loop)
    Generate one clause C
    Remove from E^+ the e^+ covered by C
    Add C to T

Generate one clause C (Specialization loop):
Select a predicate p that must be learned
Set clause C to be $p(\overline{X}) \leftarrow$.
while C covers some negative example **do**
    Select a literal L from the language bias
    Add L to the body of C
    Test coverage of C
    if C does not cover any positive example
        then backtrack to different choices for L
return C
(or fail if backtracking exhausts all choices for L)

In order to learn multiple predicates with a top-down system, two approaches are possible. We can either iteratively perform a single predicate learning task, i.e., learn completely the definition of a predicate before learning the next one, or interleaving the learning of clauses for different predicates.

The first approach suffers from the problems that have been highlighted in [5]. First, the order in which predicates are learned is relevant: for some orders it may be impossible to find a solution and thus, in the worst case, all possible orders must be considered. Second, an overgeneral definitions for a predicate p can prevent the system from finding a definition for another predicate q that depend on p. Third, it is not possible to learn mutually recursive predicates by learning completely one predicate after another.

With the second approach, i.e., by interleaving the learning of clauses for different predicates, it is possible to learn mutually recursive clauses but another problem arises: the addition of a clause that is consistent with the negative examples of its head predicate to a theory (*hypothesis*) can make it inconsistent. In this case, we have to distinguish between two types of consistency of a clause: *local* and *global consistency* of a new clause with respect to a hypothesis. These definitions slightly modify those given in [5], that were not referred to the current hypothesis. We first give some terminology and then we give the definitions of local and global consistency.

Let the training set be $E = E^+ \cup E^-$ where E^+ is the set of positive examples and E^- is the set of negative examples. We assume that E contains examples for m target predicates $p_1, \ldots, p_m$ and we partition E^+ and E^- in $E^+_{p_i}$ and $E^-_{p_i}$ according to these predicates. The *hypothesis* H is a set of clauses for some or all of the target predicates. Given the background theory B, the hypothesis H

and the example set E, the function $covers(B,H,E)$ gives the set of examples covered by H, i.e., $covers(B,H,E) = \{e \in E \mid B \cup H \vdash e\}$.

Definition 1 (Global consistency). *Given a consistent hypothesis H, clause c is* globally consistent *with respect to H if and only if $covers(B, H \cup \{c\}, E^-) = \emptyset$.*

Definition 2 (Local consistency). *Given a consistent hypothesis H, clause c for the predicate p_i is* locally consistent *with respect to H if and only if $covers(B, H \cup \{c\}, E^-_{p_i}) = \emptyset$.*

When learning multiple predicates, adding a locally consistent clause to a consistent hypothesis can produce a globally inconsistent hypothesis as it is shown in the next example inspired to [5].

Example 1. We want to learn the definitions of $ancestor$ and $father$ from the knowledge base:
$B = \{parent(a,b), parent(b,c), parent(d,b), male(a), female(b)\}$
and the training set:
$E^+ = \{ancestor(a,b), ancestor(b,c), ancestor(d,c),$
$father(a,b)\}$
$E^- = \{ancestor(b,b), ancestor(d,a), ancestor(c,b), father(b,c), father(a,c)\}$
Suppose that the system has first generated the rules:
$ancestor(X,Y) \leftarrow parent(X,Y).$
$father(X,Y) \leftarrow ancestor(X,Y), male(X).$
Clearly the second rule is incorrect but the system has no mean of discovering it now, since it is locally and globally consistent with respect to the partial definition for $ancestor$.
Then, the system learns the recursive rule for $ancestor$:
$ancestor(X,Y) \leftarrow parent(X,Z), ancestor(Z,Y).$
This clause is locally consistent with respect to the current hypothesis because none of the negative examples for $ancestor$ are covered, but it is not globally consistent because in the new theory the negative example $father(a,c)$ is now covered.

Thus, in intensional systems, it is not enough to check the consistency of a clause with respect to the negative examples for its head predicate but the consistency with respect to examples for all target predicates must be checked, as it is done in the system MPL [5]. Moreover, if a global inconsistency is found, the clauses causing it must be identified and retracted.

In order to avoid the problem of testing all negative examples and of retracting clauses, many top-down ILP systems use extensional coverage.

Definition 3 (Extensional coverage). *Given the background theory B and the example e belonging to the training set E, the clause $c = l \leftarrow l_1, l_2 \ldots l_n$* extensionally covers *$e$ iff l unifies with e with substitution θ and $B \cup E^+ \vdash [l_i]\theta$ for $i = 1 \ldots n$.*

Extensional coverage makes the evaluation of a clause independent from previously learned clauses. The system uses the training set but not the current partial hypothesis in the derivation of examples, so generated clauses are tested independently from each other. Therefore, extensional coverage avoids the problem of global inconsistency when learning multiple predicates. We no longer need to backtrack on clause addition and to search in the space of possible programs, but it is sufficient to iteratively search in the smaller space of possible clauses.

In fact, by using extensional coverage, if in example 1 the atom $ancestor(a, c)$ is included in E^+, the second rule would not be generated because $ancestor(a, c)$ would be used in testing the negative examples for $father$ and $father(a, c)$ would be covered.

However, extensional coverage poses a number of other problems: learned theories can be both inconsistent and incomplete. This is due to the fact that the extensional test is not equivalent to the intensional one. In particular, for definite logic programs, a learned theory can be [5]: (i) extensionally consistent but intensionally inconsistent, (ii) intensionally complete but extensionally incomplete or (iii) extensionally complete but intensionally incomplete (see [5] for examples of these cases).

3 Abductive Logic Programming

In this section, we summarize the main concepts of Abductive Logic Programming (ALP) that are needed for describing the algorithm.

We first give the definition of Abductive Logic Program [9].

Definition 4 (Abductive Logic Program). *An* abductive logic program *is a triple* $\langle P, A, IC\rangle$ *where*

- P *is a normal logic program,*
- A *is a set of* abducible predicates *(or* abducibles*),*
- IC *is a set of integrity constraints in the form of denials, i.e.:*
  $\leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_{m+n}.$

Abducible predicates are used to model incompleteness: these are predicates for which a definition may be missing or for which the definition may be incomplete. These are the predicates about which we can make assumptions in order to explain the current goal. More formally, given an abductive program $AT = \langle P, A, IC\rangle$ and a formula G, the goal of abduction is to find a (possibly minimal) set of ground atoms Δ (*abductive explanation*) for predicates in A which together with P entails G, i.e. $P \cup \Delta \models G$. It is also required that the program $P \cup \Delta$ is consistent with respect to IC, i.e. $P \cup \Delta \models IC$. We say that AT *abductively entails* e ($AT \models_A e$) when there exists an abductive explanation for e from AT. We adopt the three-valued semantics for ALP defined in [3] in which an atom can be *true, false* or *unknown*.

Negation as Failure is replaced, in ALP, by Negation by Default and is obtained, through abduction, in this way: for each predicate symbol p, a new predicate symbol not_p is added to the set A and the integrity constraint:

$\leftarrow p(\boldsymbol{X}), not\_p(\boldsymbol{X})$
is added to IC, where $\boldsymbol{X}$ is a tuple of variables.

Operationally, we rely on the proof procedure defined by Kakas and Mancarella [10]. This procedure starts from a goal and a set of abduced literals Δ_{in} and results in a set of consistent assumptions Δ_{out} (*abduced literals*) such that $\Delta_{out} \subseteq \Delta_{in}$ and Δ_{out}, together with the program, allows to derive the goal. In this case we write:

$$AT \vdash^{\Delta_{out}}_{\Delta_{in}} G$$

The proof procedure consists of two parts: an abductive and a consistency phase. Basically, the abductive phase differs from a standard Prolog derivation when the literal to be reduced is abducible. First, it checks to see if the abducible literal has already been assumed (i.e., it is in the Δ set) and in this case the literal is reduced. If the opposite of the literal is in Δ, the derivations fails. If the literal has not yet been abduced, the procedure tries to abduce it and checks whether it is consistent with the integrity constraints and with the current Δ by adding it to Δ and by starting a consistency derivation.

The first step of the consistency derivation consists in finding all the integrity constraints (denials for simplicity) which contain the literal. The literal can be assumed provided that all these constraints are satisfied. A denial is violated only if all its conjuncts are true, therefore at least one conjunct of each constraint must be false. Since one wants to assume the literal as true, the algorithm removes it from the constraints and checks that all the remaining goals fail. The goals are reduced literal by literal: if a literal is abducible, first it is checked if the literal itself is already in Δ (in that case the literal is dropped) or if its opposite is in Δ (in that case the constraint is satisfied and is dropped). If the literal is not in Δ, an abductive derivation for its opposite is started, so that, if this derivation succeeds, the constraint is satisfied.

In the learning algorithm we propose, negative examples will be tested by starting an abductive derivation for the negation of the example. Let us show with an example the behaviour of the procedure for the case of negative goals. Consider the following theory, inspired by [18]:

$grass\_is\_wet \leftarrow rained\_last\_night$
$grass\_is\_wet \leftarrow sprinkler\_was\_on$
$shoes\_are\_wet \leftarrow grass\_is\_wet$

Where the abducible predicates are $rained\_last\_night, sprinkler\_was\_on$ and their negation. For the goal *not_shoes_are_wet*, the procedure returns the abductive explanation

$$\Delta = \{not\_rained\_last\_night, not\_sprinkler\_was\_on\}.$$

These assumptions can be interpreted as expressing the fact that $rained\_last\_night$ and *sprinkler_was_on* must be false for *shoes_are_wet* to be false. They thus represent a set of sufficient conditions that ensure that the goal *shoes_are_wet* is not derivable in the theory.

4 The algorithm MPL-A

The algorithm MPL-A (figures 1, 2, 3) is based on the systems for learning abductive logic programs that have been presented in [11, 12]. These systems, in turn, extend the basic top-down ILP algorithm by substituting the Prolog proof procedure with the abductive proof procedure for the coverage test of examples. Therefore, a clause is tested by starting an abductive derivation for each positive example and for the default negation of each negative one. Each example can be covered or uncovered by making some assumptions. The assumptions made are collected in a set named Δ.

In order to learn multiple predicates and maintain the consistency of the learned hypothesis, the target program is considered as an abductive theory where the negation of each target predicate is an abducible predicate. Abduction is used to test the default negation of negative examples, making negative assumptions ensuring that they are uncovered. These assumptions are then added to the training set as negative examples, so that new clauses can be tested against them.

The algorithm is therefore based on a dynamic set of training examples E_c that contains the original training examples together with those generated through abduction. It rests on the important observation that, for definite logic programs, we can detect the local or global consistency of a clause by testing the training examples for its head predicate as follows:

- a clause is **locally consistent** if it does not cover any negative example from the **original training set**, while
- a clause is **globally consistent** if it does not cover any negative example from the abductively **extended training set**.

To illustrate this, consider two predicates p and q, where q depends on p. Suppose that, when testing a rule for q, an assumption $not\_p(\boldsymbol{t_p})$ for p is generated for uncovering the negative example $q(\boldsymbol{t}_q)$ for q. The assumption $not\_p(\boldsymbol{t_p})$ is then turned into the negative example $p(\boldsymbol{t_p})$ for p. Afterwards, if we learn a clause for p that covers $p(\boldsymbol{t_p})$, then also $q(\boldsymbol{t}_q)$ will be covered and the clause for p will be globally inconsistent.

Therefore, the global consistency of a clause depends only on the coverage of abduced negative examples. In procedure Evaluate (figure 3), we test negative examples with the abductive proof procedure, while positive examples are tested with the Prolog procedure, since we are interested only in negative assumptions that prohibit the coverage of some negative examples.

The procedure GenerateRule (specialization loop, figure 2) performs a beam search in the space of possible clauses. The beam is initialized with a clause with an empty body for every target predicate. Then, an heuristic function (procedure Evaluate) is used in order to select the next clause to refine. In this way, the choice of which predicate to learn next is left to the heuristic function: it will select the predicate whose clauses in the beam with the higher value for the function.

```
procedure MPL-A(
    inputs : E+, E- : training sets,
        B : background theory,
    outputs : H : learned theory, Ec \ E : abduced examples)

H := ∅
Ec := E+ ∪ not_E-
while Ec+ ≠ ∅ do (covering loop)
    GenerateRule(B, H, Ec; r, Er+, Er-, Δr)
    H := H ∪ {r}
    Ec := Ec \ Er+
    Ec := Ec ∪ Δr
    if Er- ≠ ∅ then
        RetractClauses(H, Er-, Ec; H, Ec)
    endif
endwhile
output H, Ec \ E
```

Fig. 1. Covering loop

The heuristic function is a weighted classification accuracy. The weight is given by the relative frequency of positive examples covered by the clause over the total number of positive examples in the training set and is used in order to take into account the number of positive examples covered by the rule. In fact, accuracy alone could favour very specific and accurate clauses over more general but less accurate clauses, thus possibly leading to learning theories composed by many overspecific clauses.

The procedure GenerateRule looks for a globally consistent clause that covers at least one positive example. The procedure also checks the local consistency of every refinement and stores the best one found so far. Therefore, if no globally consistent clause can be found (i.e., the beam becomes empty) but a locally consistent clause has been found, then the procedure returns it together with a non-empty set E_r^- of covered abduced negative examples. In both cases, the clause is added to the theory and the negative assumptions generated when testing the clause are added to the training set. Then, if the clause is only locally consistent, backtracking on previous clauses is performed. If no locally consistent solution exists that covers at least one positive example, the algorithm fails.

Backtracking (procedure RetractClauses in figure 1) is performed by retracting the clauses that have generated the negative examples covered by the locally consistent clause, i.e., those that contained in the body the corresponding abducible literal, since they are made inconsistent by the addition of the new one. These clauses are retracted, positive examples covered by them are re-added to the training set and the negative examples generated by them are removed from the training set. In order to perform backtracking, the system has to store, for each assumption, the clause that has generated it. Each retracted clause is then

procedure GenerateRule(
  inputs : B : background theory,
    H : current hypothesis, E_c : training set,
  outputs : $Best$: rule,
    E^+_{Best}, E^-_{Best} : positive and negative examples covered by $Best$,
    Δ_{Best} : assumptions generated by $Best$

$Beam := \{ \langle p(X) \leftarrow true., Value \rangle \quad | \quad p$ is a target predicate,
  $Value := \frac{|E^+_p|}{|E^+_p|+|E^-_p|} \}$
$LocallyConsClause := nil$
repeat
  remove the $Best$ rule from $Beam$
  $BestRefinements :=$ set of refinements of $Best$ allowed
    by the language bias
  for all $Rule \in BestRefinements$ **do**
    Evaluate($Rule, B, H, E_c; Value, E^+_{Rule}, E^-_{Rule}, \Delta_{Rule}$)
    if $Rule$ covers at least one positive example **then**
      add $\langle Rule, Value \rangle$ to $Beam$
      if $Rule$ is locally consistent and
      $Rule$ is better than $LocallyConsClause$ **then**
        $LocallyConsClause := Rule$
      endif
    endif
  endfor
  remove the rules in $Beam$ exceeding the $Beamsize$
until the $Best$ rule in $Beam$ is globally consistent or the $Beam$ is empty
if no globally consistent clause can be found ($Beam$ is empty) **then**
  if $LocallyConsClause$ is not nil **then**
    $Best := LocallyConsClause$
  else
    fail
  endif
endif
Evaluate($Best, B, H, E_c; Value, E^+_{Best}, E^-_{Best}, \Delta_{Best}$)
output $Best, E^+_{Best}, E^-_{Best}, \Delta_{Best}$

Fig. 2. Specialization loop

procedure Evaluate(
  inputs : $Rule$: rule, B : background theory,
    H : current hypothesis, E_c : training set,
  output : $Value$: the value of the heuristic function for $Rule$,
    E^+_{Rule}, E^-_{Rule} : examples covered by $Rule$
    Δ_{Rule} : new set of abduced examples)

n^+ := covered positive examples (tested with the Prolog proof procedure)
n^- := 0, number of negative examples
$\Delta_{Rule} := \emptyset$
for each $e^- \in E_c$ **do**
  AbductiveDerivation($not\ e^-, \langle B \cup H \cup \{Rule\}, A, I\rangle, E_c; \emptyset, \Delta_{e^-}$)
  if the derivation succeeds **then**
    $\Delta_{Rule} := \Delta_{Rule} \cup \Delta_{e^-}$
  else
    increment n^-
  endif
endfor
$Value := \frac{n^+}{|E_c^+|} \times \frac{n^+}{n^+ + n^-}$
return $Value, \Delta_{Rule}$

Fig. 3. Clause evaluation

added to a list of retracted clauses so that it can not be added anymore to the theory: in the case in which it is generated again in the specialization loop, it is immediately discarded. This is done in order to avoid that the system goes into a loop of continuously generating and retracting the same clause.

Finally, in the abductive proof procedure, we consider examples of other target predicates as background facts, thus obtaining a hybrid extensional-intensional system. Being a hybrid system, it does not incur in two of the problems of extensional systems, namely extensional consistency, intensional inconsistency and intensional completeness, extensional incompleteness. On the other hand, it can incur in the third, i.e., extensional completeness, intensional incompleteness. For example, it can learn two mutually recursive clauses that intensionally lead to a loop while cover extensionally the examples. Subject for future work is to extend the system with the techniques proposed in [16] for learning recursive predicates.

By means of the hybrid coverage adopted, the system is less sensitive to the order of learning the predicates because it can exploit examples for defining the predicates that it has not yet learned. In this way, possible dead-ends of the search can be detected in advance.

5 Experiments

In this section we present some experiments that have been performed with the MPL-A system: learning the definition of *father* and *ancestor* from the data in example 1, learning the definition of $father$ and $grandfather$ and learning a definite clause grammar for simple sentences.

5.1 Father and Ancestor

We now show the behaviour of the system in the case of examples 1.

When the system tests the rule

$father(X,Y) \leftarrow ancestor(X,Y), male(X)$

it generates the assumptions

$\{not\_ancestor(b,c), not\_ancestor(a,c)\}$

that become negative examples for ancestor. When it tries to learn the recursive clause for ancestor, it will not be able to find a clause that is consistent with $not\_ancestor(a,c)$, therefore it will generate the locally consistent clause

$ancestor(X,Y) \leftarrow parent(X,Z), ancestor(Z,Y)$

and it will retract the clause for father that has generated the covered negative example $not\_ancestor(a,c)$. At this point, the correct rule for $father$ can be learned.

5.2 Father and Grandfather

We want to learn the predicates $grandfather$ and $father$ from the background theory:

$P = \{parent(john, steve), male(john), male(steve)$
$parent(steve, ellen), female(ellen)$
$parent(ellen, sue), female(sue)\}$

and the training set:

$E^+ = \{grandfather(john, ellen), grandfather(steve, sue),$
$\quad father(john, steve)\}$
$E^- = \{grandfather(mary, sue), father(john, ellen)\}$

MPL-A learns first the rule for $grandfather$ because the heuristic function prefers it to any of the rules for $father$. When MPL-A generates the rule

$grandfather(X,Y) \leftarrow parent(Z,Y), father(X,Z).$

it uses the examples for $father$ as background knowledge making also negative assumptions about it when this is needed. Given the training examples for $grandfather$

$E^+_{gf} = \{grandfather(john, ellen), grandfather(steve, sue)\}$
$E^-_{gf} = \{grandfather(mary, sue)\}$

M-ACL will produce, together with the above rule, the following assumption:

$\{not\_father(mary, ellen).\}$

This become an additional training example for $father$. From this new training set, the system is then able to generate the correct rule for $father$. Note that without the new negative example $father(mary, ellen)$ it would have been impossible to generate the correct rule for $father$ and the overgeneral rule $father(X,Y) \leftarrow parent(X,Y)$ would have been learned. Thus MPL-A is able to avoid (in this case) the problem of overgeneralization.

5.3 Grammar

The data for this experiment is taken from [4]. The aim is to learn the following definite clause grammar for parsing very simple English sentences:

(1) $sent(A, B) \leftarrow np(A, C), vp(C, B)$.
(2) $np(A, B) \leftarrow det(A, C), noun(C, B)$.
(3) $vp(A, B) \leftarrow verb(A, B)$.
(4) $vp(A, B) \leftarrow verb(A, C), np(C, B)$.

In [4] Claudien-Sat is used to solve this task starting from different input interpretations.

The first interpretation corresponds to a complete syntactic analysis of the sentence "the dog eats the cat". Therefore the data set contains all the positive and negative facts mentioning the following lists: [the,dog,eats,the,cat], [dog,eats,the,cat], [eats,the,cat], [the,cat], [cat] and []. Another interpretation contains some ungrammatical sentences and corresponds to several attempts to analyze "the cat the cat". It includes all positive and negative facts mentioning the following lists: [the,cat,the,cat], [cat,the,cat], [cat,cat], [the,cat], [cat], [cat,the] and []. Similarly, another interpretation contains all positive and negative facts mentioning the lists [the,cat,eats], [cat,eats], [cat,sings], [the,cat,sings], [dog,cat], [sings], [eats],[the] and [].

M-ACL has learned the above rules in the following order: (2), (3), (1), (4). Note that the definition for *sent* was learned at a point where the definition for *vp* was not complete. This was possible because the system used the examples for *vp* to complete its definition, by exploiting the hybrid form of coverage. Some negative assumptions about *np* were made in order to avoid the coverage of negative examples.

6 Related Work

This paper is based on the work on learning abductive logic programs in [11, 12]. The systems presented in these papers are modified and extended in order to apply them to the problem of learning multiple predicates: abduction is used only for the coverage of test of negative examples, local and global consistency are distinguished and backtracking on clause addition is performed.

On the problem of learning multiple predicates a notable work is [5] where the authors thoroughly analyze the problem and the solutions proposed both by intensional and extensional systems. In order to overcome the problem of global inconsistency for intensional systems, they propose the system MPL that takes a different approach with respect to ours. After the addition of each clause, in order to detect a global inconsistency in the current hypothesis, it retest the hypothesis against all negative examples. On the contrary, we are able to detect the global inconsistency by testing only a limited number of negative examples.

Our system still suffers from the problem of extensional completeness, intensional incompleteness. This problem has been deeply studied, both for definite and normal logic program, in [16]. The authors propose the system ICN in which they solve the problem by keeping explicit track of the recursive dependency among clauses. An interesting direction for future work would be to incorporate their solution into our system.

Hybrid coverage is used as well in the system FOIL-I [8]. There the authors especially concentrate on learning recursive predicates from a sparse training set and they do not investigate the properties of such a system with respect to multiple predicate learning.

Since we gradually add negative examples, our approach may seem similar to the one adopted in incremental systems such as MIS [20]. However, while in incremental systems a consistency check must be done after the addition of each e^- to the training set, we do not need to do this because we add an e^- only after having tested that it is not covered by any clause.

7 Conclusions and Future Work

We have shown how abduction can be used to overcome the problem of global inconsistency when learning multiple predicates without incurring in the problems of extensional systems, apart from the one of extensional completeness, intensional incompleteness.

This work was inspired by [11–13] where the (intensional) algorithm for learning abductive logic programs was introduced and its main properties studied. We improve on that work by adding a mechanism for detecting global inconsistency and for performing clause backtracking.

In the future, we will investigate the application of similar techniques to the problem of learning logic programs with negation (*normal logic programs*). In this case, the addition of a clause to a hypothesis can reduce the coverage of the hypothesis, thus making impossible to use a standard covering algorithm. With abduction we are able to generate additional examples that can be used to avoid this problem.

References

1. H. Adé and M. Denecker. AILP: Abductive inductive logic programming. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 1995.
2. F. Bergadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering.* The MIT Press, 1995.
3. A. Brogi, E. Lamma, P. Mancarella, and P. Mello. A unifying view for logic programming with non-monotonic reasoning. *Theoretical Computer Science*, 184:1–59, 1997.
4. L. De Raedt and L. Dehaspe. Learning from satisfiability. Technical report, Katholieke Universiteit Leuven, 1996.
5. L. De Raedt, N. Lavrač, and S. Džeroski. Multiple predicate learning. In S. Muggleton, editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 221–240. J. Stefan Institute, 1993.

6. M. Denecker, L. De Raedt, P. Flach, and A. Kakas, editors. *Proceedings of ECAI96 Workshop on Abductive and Inductive Reasoning.* Catholic University of Leuven, 1996.
7. Y. Dimopoulos and A. Kakas. Abduction and inductive learning. In *Advances in Inductive Logic Programming.* IOS Press, 1996.
8. N. Inuzuka, M. Kamo, N. Ishii, H. Seki, and H. Itoh. Top-down induction of logic programs from incomplete samples. In S. Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming*, number 1314 in LNAI, pages 265–284. Springer-Verlag, 1997.
9. A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in AI and Logic Programming*, volume 5, pages 233–306. Oxford University Press, 1997.
10. A.C. Kakas and P. Mancarella. On the relation between truth maintenance and abduction. In *Proceedings of the 2nd Pacific Rim International Conference on Artificial Intelligence*, 1990.
11. A.C. Kakas and F. Riguzzi. Learning with abduction. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, 1997.
12. E. Lamma, P. Mello, M. Milano, and F. Riguzzi. Integrating induction and abduction in logic programming. To appear on Information Sciences.
13. E. Lamma, P. Mello, M. Milano, and F. Riguzzi. Integrating extensional and intensional ILP systems through abduction. In *Proceedings of the 7th International Workshop on Logic Program Synthesis and Transformation*, 1997.
14. E. Lamma, P. Mello, M. Milano, and F. Riguzzi. Integrating Induction and Abduction in Logic Programming. In P. P. Wang, editor, *Proceedings of the Third Joint Conference on Information Sciences*, volume 2, pages 203–206, 1997.
15. L. Martin and C. Vrain. MULT_ICN: An empirical multiple predicate learner. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pages 129–144. Department of Computer Science, Katholieke Universiteit Leuven, 1995.
16. L. Martin and C. Vrain. A three-valued framework for the induction of general program. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pages 109–128. Department of Computer Science, Katholieke Universiteit Leuven, 1995.
17. M.J. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94, 1992.
18. J. Pearl. Embracing causality in formal reasoning. In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 369–373, Seattle, WA, 1987.
19. J. R. Quinlan and R.M. Cameron-Jones. Induction of Logic Programs: FOIL and Related Systems. *New Generation Computing*, 13:287–312, 1995.
20. E. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.