Integrating Induction and Abduction in Logic Programming

Evelina Lamma [1] Paola Mello [2] Michela Milano [1] Fabrizio Riguzzi [1]

[1] *DEIS, Università di Bologna*
Viale Risorgimento 2, 40136 Bologna, Italy
Tel: +39 51 6443033, Fax: +39 51 6443073
{elamma,mmilano,friguzzi}@deis.unibo.it

[2] *Istituto di Ingegneria, Università di Ferrara*
Via Saragat 1, 41100 Ferrara, Italy
pmello@ing.unife.it

1 Introduction

We propose an approach for the integration of inductive and abductive reasoning. Both abduction and induction have been recognized as powerful mechanisms for hypothetical reasoning in presence of incomplete knowledge [4, 7, 9, 11, 12]. Abduction is generally understood as reasoning from effects to causes or explanations. Given a theory T and a formula G, the goal of abduction is to find a (possibly minimal) set of atoms Δ which together with T entails G. Induction is generally understood as inferring general rules from specific data. Given a theory T and a formula (observation) G, the goal of induction is to find a set of general rules Δ (of the type $\alpha \rightarrow \beta$) which together with T entails G.

In this paper, we concentrate on Logic Programming. In particular, we extend an Inductive Logic Programming (ILP, for short [3]) algorithm in order to manipulate abductive logic programs. An ILP problem can be defined as follows: given a set $\mathcal{P}$ of possible programs, sets E^+ and E^- of positive and negative examples, a consistent logic program B (background knowledge), find a logic program $P \in \mathcal{P}$ such that $B \cup P$ entails (or covers) E^+ and does not entail E^-. In this work, we propose a general approach where it is possible to learn, by induction, an abductive logic program [7, 8], i.e., a logic program P (possibly with abducible atoms in clause bodies), a set of abducibles $\mathcal{A}$ and a set of integrity constraints IC. Abducibles are predicates which can be assumed true (or false) during the computation provided that they are consistent with integrity constraints. Therefore, they can be used to deal with incomplete information.

The inductive algorithm here presented is an extension of a top-down algorithm adopted in ILP [3]. The extended algorithm takes into account abducibles and integrity constraints, and is intertwined with the proof procedure defined in [10] for abductive logic programs.

The problem of integrating ILP with ALP has been investigated by several researchers in Artificial Intelligence, see for instance [1, 2, 6]. The main advantages of the integration concern the increased expressive power of the learned program, and the possibility of learning in presence of incomplete knowledge. We can take into account user-defined abducibles, and introduce new abducibles and integrity constraints in order to generate exceptions (as in [5]) to induced rules.

2 Preliminaries on ILP and ALP

In this section, we first briefly recall some basic concepts of Abductive Logic Programming (ALP). Then we describe, at a high level, a basic Inductive Logic Programming (ILP) algorithm.

In the context of abduction, missing information is represented by (user-defined) abducible predicates, possibly constrained by integrity constraints. An *abductive logic program* is a triple $\langle P, \mathcal{A}, IC \rangle$ where:

- P is a normal logic program, that is, a set of clauses of the form $A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_{m+n}$, where $m, n \geq 0$ and each A_i $(i = 1, \ldots, m+n)$ is an atom;
- $\mathcal{A}$ is a set of *abducible predicates* comprehensive of any default literal $not\ A$;
- IC is a set of integrity constraints (denials, for simplicity).

In [10] a proof procedure for abductive logic programs has been defined. This procedure starts from a goal and results in a set of consistent hypothesis (abduced literals) that together with the program allow to derive the goal.

Now we define some basics on ILP. Starting from a background knowledge and from sets of positive and negative examples, an ILP algorithm learns a set of clauses that cover the positive examples and does not cover the negative ones. A basic top-down inductive algorithm [3] learns programs by generating clauses one after the other. Each clause is successively refined,

while T entails some $e^+ \in E^+$ do
    Generate one clause C
    Remove from E^+ the e^+ covered by C
    Add C to T

Generate one clause C:
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
    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)

Figure 1: Basic ILP algorithm

starting from an empty body, by adding further literals to the body. Let T denote the set of induced clauses, initially empty, and E^+ and E^- be the training set (positive and negative examples, respectively). The basic inductive algorithm is sketched in figure 1.

In this paper, we propose an extension of the ILP algorithm which is informally described in the next section by means of examples.

3 Examples

The purpose of our framework is (in analogy with [6]) to synthesize a new abductive theory $\langle P', \mathcal{A}', IC' \rangle$, starting from an abductive logic program $\langle P, \mathcal{A}, IC \rangle$ and a set of positive and negative observations for a concept c. The new theory contains rules for the concept c, and new abducibles and integrity constraints are possibly introduced in order to cope with exceptions and rule specialization. In the following we present, by means of examples, the intended behavior of our system.

3.1 Learning Rules with Exceptions

The first example is inspired to [5]. Let us consider the following background knowledge P:

$bird(X) \leftarrow penguin(X)$.
$penguin(X) \leftarrow superpenguin(X)$.
$bird(a)$.
$bird(b)$.
$penguin(c)$.
$penguin(d)$.
$superpenguin(e)$.
$superpenguin(f)$.

and the set of examples:

$E^+ = \{flies(a), flies(b), flies(e), flies(f)\}$
$E^- = \{flies(c), flies(d)\}$

Let $\mathcal{A}$ and IC be the empty set. The algorithm generates the following rule (R_1):

$flies(X) \leftarrow superpenguin(X)$.

and removes $flies(e)$ and $flies(f)$ from E^+. Then, rule generation produce the following rule (R_2):

$flies(X) \leftarrow bird(X)$.

which covers all the remaining positive examples, but also the negative ones. In fact, the abductive derivations for $not\ flies(c)$ and $not\ flies(d)$ fail. There is no way to specialize the generated rule by introducing (user-defined) abducibles. In order to rule out negative examples, a new abducible literal is generated, $not\ abnorm_1$, and added to the body of R_2. Moreover, the integrity constraint:

$\leftarrow abnorm_1(X), not\ abnorm_1(X)$.

is also added to the background knowledge. Now, the abductive derivation for $not\ flies(c)$ (respectively, for $not\ flies(d)$) succeeds, provided that the abducible $abnorm_1(c)$ (resp. $abnorm_1(d)$) is assumed true. Intuitively, at this point, we could just add these facts to the learned program, or better, try to generate a rule for them, as if they were considered as new positive examples to be covered for predicate $abnorm_1$. The negative examples for this new concept to be learned correspond to the positive ones (i.e., $abnorm_1(a)$, $abnorm_1(b)$ which correspond to the elements of E^+). The resulting induced rule is (R_3):

$abnorm_1(X) \leftarrow penguin(X)$.

At this point, via rule R_2 and R_3 all the remaining positive examples are covered, whereas the negative ones are uncovered. Thus, E^+ becomes empty, and the algorithm ends by producing the following abductive program, where P' is:

$flies(X) \leftarrow superpenguin(X)$.
$flies(X) \leftarrow bird(X), not\ abnorm_1(X)$.
$abnorm_1(X) \leftarrow penguin(X)$.

the set of abducibles is $\mathcal{A}' = \mathcal{A} \cup \{not\ abnorm_1\}$ and the set of integrity constraints:

$IC' = IC \cup \{\leftarrow not\ abnorm_1(X), abnorm_1(X)\}$

An equivalent program for the same example is obtained in [5], but exploiting negation rather than abduction. However, in [7] the authors have argued that negation *by default* can be seen as a special case of abduction. The power of negation *by default* in induction is preserved and, what is more, is generalized by abduction. Thus, by integrating induction and abduction, we can achieve greater generality with respect to [5]. Nonetheless, the treatment of exceptions here adopted is very similar to that introduced in [5] through a limited form of "classical" negation and priority relations between rules.

3.2 Learning from Integrity Constraints

In previous example, the background knowledge is very simple since the program P does not contain abducibles and integrity constraints defined by the user. As a further example, let us consider the abductive program $\langle P', \mathcal{A}', IC' \rangle$ generated in previous subsection, and add to it the following constraint, I, defined by the user:

$\leftarrow rests(X), plays(X).$

Consider now the new training set:

$E^+ = \{plays(a), plays(b), rests(e), rests(f)\}$

$E^- = \{\}$

Notice that the added integrity constraint I can be mapped into the following (non-empty) set of negative examples:

$E^- = \{rests(a), rests(b), plays(e), plays(f)\}$

The inductive algorithm first generates the following rule:[1]

$plays(X) \leftarrow bird(X).$

which covers all the positive examples for *plays*, but also the negative examples $plays(e)$ and $plays(f)$. In practice, the generated rule violates the original integrity constraint.

Differently from example of section 3.1, in order to specialize the rule and possibly restore consistency, we can exploit the abducible predicate $not\ abnorm_1 \in \mathcal{A}'$, and add it to the body of the generated rule:

$plays(X) \leftarrow bird(X), not\ abnorm_1(X).$

Now the abductive derivations for $not\ plays(e)$ and $not\ plays(f)$ succeeds since both $abnorm_1(e)$ and $abnorm_1(f)$ can be derived. This suffices for ruling out the negative examples for *plays*. Therefore, the predicate *plays* holds only for one subclass of the class *bird*, i.e., only for birds which are not penguins.

When iterating, the algorithm also generates a rule for *rests*:

$rests(X) \leftarrow superpenguin(X).$

In this way, we have increased the power of the learning process. We can learn not only by (positive and negative) examples but also by integrity constraints, like in [13].

3.3 Learning Integrity Constraints

Let us analyze another case. Suppose we have an empty background knowledge, no abducible and no integrity constraint. Suppose also we have the following positive and negative examples:

$E^+ = \{plays(a), rests(b)\}$

$E^- = \{plays(b), rests(a)\}$

A standard inductive algorithm cannot infer any general information, since the fact:

$plays(X) \leftarrow .$

covers the negative example $plays(b)$, and the fact:

$rests(X) \leftarrow .$

covers the negative example $rests(a)$. This problem clearly derives from the universal quantification. The facts inferred are too general, and the only way in which they can be specialized is by adding the facts:

$plays(a) \leftarrow .$

$rests(b) \leftarrow .$

which are too specific. Therefore, we can think of an intermediate way of learning from these examples. By induction, our algorithm first generates the rule:

$plays(X).$

that also covers the negative example $plays(b)$ (the abductive derivation for $not\ plays(b)$ being a failure). In order to restore consistency, a new (abducible) predicate is generated, $not\ abnorm_1$, and added to the body of the rule:

$plays(X) \leftarrow not\ abnorm_1(X).$

Moreover, the following integrity constraint is generated:

$\leftarrow abnorm_1(X), not\ abnorm_1(X).$

Now, the abductive derivation for $not\ plays(b)$ succeeds provided that $abnorm_1(b)$ is assumed. At this point, we could just add this fact to the learned program, or better, try to generate a rule for it and generalize. Differently from example in section 3.1, however, rule generalization would lead to a loop. In fact, by following the proposed algorithm, $abnorm_1(b)$ would be generalized by introducing a new (default) abducible $not\ abnorm_2$, thus leading to a loop. Therefore, to avoid loop, we prefer to restrict the algorithm and avoid to generate a rule for an abducible predicate containing only abducible predicates in its body, and define $abnorm_1$ as:

$abnorm_1(b) \leftarrow .$

Then, the algorithm generates a rule for *rests*:

$rests(X) \leftarrow abnorm_1(X).$

and terminates. It is matter of discussion whether it is convenient to generalize the induced integrity constraint or not. By generalizing the integrity constraint:

$\leftarrow abnorm_1(X), not\ abnorm_1(X).$

one obtains the constraint:

$\leftarrow plays(X), rests(X).$

This integrity constraint is quite meaningful. In fact, even if there is no relation between a and b, and we do not have information on them, we can observe that the properties *plays* and *rests* are somehow contradictory. In our world, in fact, there is nothing that has both these properties. By induction and generalization of the learned integrity constraint, we can infer the integrity constraint which asserts that it is not possible that an object X in our world both *plays* and *rests*.

[1] We will analyze the case of *rests* later.

4 The Basic Algorithm

The basic inductive algorithm, presented in section 2, is here extended in the following respects.

First, when clause C is generated in order to cover positive examples, its body might also contain abducibles, in analogy with the framework in [6]. Thus, the selected literal L for the specialization can be a literal of the background knowledge, of the training set or a user-defined abducible.

Second, in order to determine the positive examples covered by the generated clause C, an *abductive* derivation [10] is started for each positive example. This procedure results in a (possibly empty) set of consistent hypothesis (abduced literals) that together with the program allow to derive the examples. Since we want the set of abducibles to be consistent for all the examples, the abductive procedure starts with a non empty set of abduced literals (assumed in the derivation of the previous examples) and extends it. In the algorithm, we have to replace the step *Remove from* E^+ *the positive examples covered by* C with the following two steps:

- keep a set of literals abduced during the testing of positive and negative examples;
- use this set (instead of an empty one) as the starting set in the abductive derivation of the remaining positive and (negated) negative examples.

As well, in order to check that no negative example is covered by the generated clause C, an *abductive* derivation is started for the negation of each negative example in E^-. If these derivations succeed, no negative example is covered. Again, the set of abduced literals is extended during the computation in order to maintain the consistency among abducibles.

Finally, when generating a clause, rather than failing if backtracking exhausts all choices for literal L, we choose to introduce new abducibles and integrity constraints. In particular, during clause generation, if the current explanation also covers some negative examples and no further literal can be added to the clause body, then a new abducible literal *not abnorm* is automatically generated and introduced in the body of the rule. Then, a new rule for predicate *abnorm* is synthesized as exception to the previous one in order to rule out the negative examples, as done in [5].

5 Conclusions and Future Work

We have discussed how it is possible to learn an abductive logic program addressing in this way nonmonotonicity and exceptions. In the devised framework, abducibles and integrity constraints can be specified by the user and are also generated by the learning process. In this way, we increase the expressive power of the background knowledge and make it possible to learn in presence of incomplete information.

We have implemented in Prolog the extension proposed by starting from the ILP basic algorithm. The implementation works correctly for the examples shown in this paper and for the examples shown in [5, 6] and gave very good results. We are currently testing the algorithm on real classification and planning problems.

The possibility of learning integrity constraints should be further investigated. In section 3.3, we have shown how to learn binary integrity constraint. How to learn general constraints is subject for future works.

References

[1] H. Adé and M. Denecker. AILP: Abductive Inductive Logic Programming. In *Proceedings IJCAI95*, 1995.

[2] M. Bain and S. Muggleton. *Non-Monotonic Learning in Inductive Logic programming*, chapter 7. Academic Press, 1992.

[3] F. Bergadano and D. Gunetti. *Inductive Logic Programming*. MIT press, 1996.

[4] L. Console, D. Theseider Duprè, and P. Torasso. *Abductive Reasoning Through Direct Deduction from Completed Domains Models in Methodologies for Intelligent Systems*. North Holland, 1989.

[5] Y. Dimopoulos and A. Kakas. Learning Non-monotonic Logic Programs: Learning Exceptions. In *Proceedings ECML95*, 1995.

[6] Y. Dimopoulos and A. Kakas. *Abduction and Learning in Advances in Inductive Logic Programming*. IOS Press, 1996.

[7] K. Eshghi and R.A. Kowalski. Abduction compared with Negation by Failure. In *Proceedings of ICLP89*, 1989.

[8] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2:719–770, 1993.

[9] A.C. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proceedings of ECAI90*, 1990.

[10] A.C. Kakas and P. Mancarella. On the relation between Truth Maintenance and Abduction. In *Proceedings of PRICAI90*, 1990.

[11] R. Michalski, J.G. Carbonell, and T.M. Mitchell (eds). *Machine Learning - An Artificial Intelligence Approach*. Springer Verlag, 1984.

[12] R. Michalski, J.G. Carbonell, and T.M. Mitchell (eds). *Machine Learning - An Artificial Intelligence Approach Vol. II*. Morgan Kaufmann, 1986.

[13] L. De Raedt and M. Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence*, 53:291–307, 1992.