Systematic Predicate Invention in Inductive Logic Programming

L. Martin, C. Vrain
LIFO - université d’Orléans
BP 6759 - 45067 Orléans cedex 02 - France
email: {martin.cv}@lifo.univ-orleans.fr

Abstract

We propose in this paper a new approach for learning predicate definitions from examples and an initial theory. The particularity of this approach consists in inventing a new predicate at most steps of learning; once the learning task is ended, most invented predicates are removed by unfolding techniques. Nevertheless, some predicates remain in the learned definitions, either because they enable to simplify the program, or because their definitions are recursive and the program could not have been learned without inventing these predicates. Moreover, when a new predicate symbol is introduced, a specification for this predicate is built; this specification is both incomplete and imprecise, what we modelize by introducing the notion of $\sigma$-interpretation. It is worth noted that the proposed method can be used, even when the target concepts are also incompletely defined by $\sigma$-interpretations.

The method has been implemented in the system, called SPILP, which has been successfully tested for inventing predicates which simplify the learned programs as well as for inventing recursively defined predicates.

Keywords: Inductive Logic Programming, predicate invention, constructive learning, noise.
1 Introduction

In this paper, we propose a new method for Constructive Learning which is based on a systematic use of predicate invention. Constructive learning is an active research topic in Machine Learning and Inductive Logic Programming (ILP) [2, 4, 7, 8, 9, 17], which is defined as the capacity to extend a fixed initial representation by adding new non-observational representation primitives [4]; in ILP, these new primitives are mainly predicate symbols. It enables to overcome the limitation of most classical approaches, as for instance Foil [14] or ICN [6], that restrict the language of hypotheses to the target predicate and to the basic predicates occurring in the initial theory \( T \).

To illustrate this point, let us consider a domain of 9 cities and 2 basic predicates \( fl \) and \( tr \) such that \( p(X,Y) \) is true iff there exists a direct flight from \( X \) to \( Y \), and \( tr(X,Y) \) is true iff there exists a train going from \( X \) to \( Y \). Let us consider the target predicate \( p \) defined by: \( p(X,Y) \) is true iff both, there exists a direct flight from \( X \) to \( Y \) and it is possible to go from \( X \) to \( Y \) by train. This concept cannot be learned without introducing a predicate \( by\_train(X,Y) \), representing the transitive closure of \( tr \), i.e. \( \), meaning that it is possible to go from \( X \) to \( Y \) by train. In other words, systems like Foil [14] or ICN [6] are unable to learn a definite logic program defining the predicate \( p \). The system SPLIP that we present here learns the program given in the last column of Table 1. The last two clauses can be simplified into \( R1(X,Y) \leftarrow tr(X,Z), R2(Z,Y) \).

<table>
<thead>
<tr>
<th>Basic predicates</th>
<th>Target predicate</th>
<th>Learned program</th>
</tr>
</thead>
<tbody>
<tr>
<td>( tr(d,a) ), ( tr(a,b) ), ( tr(b,c) ), ( tr(b,e) ), ( tr(e,f) ), ( tr(f,g) ), ( tr(h,i) ), ( fl(d,a) ), ( fl(a,c) ), ( fl(e,d) ), ( fl(b,g) ), ( fl(e,g) ), ( fl(b,c) ), ( fl(c,g) ), ( fl(h,i) )</td>
<td>( p(d,a), p(a,c) ), ( p(b,g), p(b,c) ), ( p(e,g), p(h,i) )</td>
<td>( p(X,Y) \leftarrow fl(X,Y), R1(X,Y) \cdot R1(X,Y) \leftarrow tr(X,Y) \cdot R1(X,Y) \leftarrow tr(X,Z), R2(X,Y,Z) \cdot R2(X,Y,Z) \leftarrow R1(Z,Y) ).</td>
</tr>
</tbody>
</table>

Table 1.

In our method, the learned definitions contain predicates, as for instance \( R1 \) and \( R2 \), which are invented and defined during the learning process.

We can consider two advantages within predicate invention:

**simplicity/comprehensibility**: in the best case, invented predicates correspond to predicates that were missing in the initial theory and which have a particular meaning on the domain.

**completeness**: Some learning problems can be solved only if the description language is widened by the invention of predicates. In that sense, learning with predicate invention enables to increase the completeness of the learning process.

These two advantages have already been mentioned in [4], where predicates which do not affect the learnability are called *useful* and the other ones are called *necessary*.

Two types of approaches have been proposed for predicate invention: on one hand, the systems DUCE [7] and CIGOL [9] introduce the intra-construction in the framework of inverse resolution, which allows to introduce only *useful* predicates simplifying the program. On the other hand, the CILP algorithm [4] is based on the notion of *sub-unification* of terms in the context of inverting implication; this allows to produce a particular form of recursive clauses by analysing terms containing
function symbols, CILP can then invent necessary predicates defined recursively. In this paper, we propose a new approach of predicate invention which is neither based on inverse resolution, nor on inverse implication. The approach proposes a uniform method to learn both target predicates and invented predicates. The basic operation consists in introducing a new predicate and building its specification; invented predicates are then considered as target predicates. The definitions learned for both kinds of predicates may be recursive. We do not use folding techniques to introduce new predicates, but unfolding ones in order to remove unnecessary or useless invented predicates. For the time being, our approach is restricted to learn function free logic programs.

The paper is organized as follows. Section 2 introduces the basic mechanism of our approach. Section 3 presents the system SPILP that we have developed based on this method and Section 4 gives examples of learned programs. Finally, we conclude with a discussion of the complexity and a presentation of further developments.

2 Basic mechanism

In the following, \( T \) represents an initial theory, defining a set of basic predicates (denoted by \( p_1, p_2, \ldots \)), \( I_T \) denotes a finite set of ground atoms, true in \( T \), \( R_0 \) denotes the target predicate and \( R_i, i > 0 \) denotes an invented predicate.

The specification of the target predicate is usually given by two sets of ground atoms: the set of positive examples and the set of negative ones. In this paper, we define a more general notion of a predicate specification, which enables to handle the case where the positive examples are imprecise, in the following way:

**Definition** A \( \sigma \)-interpretation is a pair \( < E^+_\sigma, E^-_\sigma > \) where \( E^- \) is a set of ground atoms, defining the negative examples and \( E^+_\sigma \) is a set \( \{ \sigma_1, \ldots, \sigma_n \} \) where each \( \sigma_i \) is a finite set of ground atoms containing at least a positive example.

Let us notice that this notion goes beyond the problem of predicate invention, addressed in this paper, since it can be compared to other learning problems in the context of incomplete specifications, such as for instance [1].

Given a \( \sigma \)-interpretation \( E = < E^+_\sigma, E^-_\sigma > \) for the target predicate, our goal is to build a program \( P \) which is consistent, i.e. no negative example is true for \( P \), and \( \sigma \)-complete, i.e. each \( \sigma_i \in E^+_\sigma \) contains an atom true for \( P \). This acceptability criterion can be reformulated as follow:

- **\( \sigma \)-completeness:** \( \forall \sigma_i \in E^+_\sigma, \exists \in \sigma_i \) such that \( P, T \models l \),
- **consistency:** \( \forall e \in E^-, P, T \not\models e \).

**Remark:** The method proposed here allows to specify the target predicate with a \( \sigma \)-interpretation. For sake of simplicity, we suppose, in the following that the initial target predicate is defined by a set \( \{ e_1, \ldots, e_n \} \) of positive examples and a set \( \{ e'_1, \ldots, e'_m \} \) of negative ones. Let us point out that this is a particular case of \( \sigma \)-interpretation \( E \) by taking \( E^+_\sigma = \{ \{ e_1 \}, \ldots, \{ e_n \} \} \) and \( E^- = \{ e'_1, \ldots, e'_m \} \).

**Notations:** \( P(\overline{Y}) \) denotes the atom \( P(Y_1, \ldots, Y_n) \) and \( P(\overline{e}) \) denotes the ground atom \( P(c_1, \ldots, c_n) \)

2.1 Introductory Example

Let us consider the basic predicates \( father \) and \( mother \) defined by the ground atoms:
\{father(a, c), father(c, e), father(a, d), mother(b, c), mother(b, d),\}

The learned predicate \(R_0 = \text{parent} \rightarrow \text{father}\) is defined by:
\[
E_{a0}^+ = \{\{R_0(a, c),\}, \{R_0(a, f),\}, \{R_0(a, g)\}\}
\]

\(\text{parent} \rightarrow \text{father}\) is defined by a set of positive examples and \(E_0^\rightarrow\) is a set of negative examples containing at least \(R_0(a, b)\) but eventually incomplete.

The learned program is initially empty. Let us assume that the first selected literal for the definition of \(R_0\) is \(\text{father}(X, Z)\). Since the clause
\[
\text{parent}(X, Y) \rightarrow \text{father}(X, Z).
\]
covers the negative example \(R_0(a, b)\), a new predicate, called \(R_1\), is invented and the clause:
\[
\text{parent}(X, Y) \rightarrow \text{father}(X, Z), R_1(X, Y, Z).
\]
is added to the program \(\mathcal{P}\). The set \(E_1^\rightarrow\) is built, in the way described in Section 2.1.2. It contains \(R_1(a, b, c)\) since \(R_0(a, b) \rightarrow \text{father}(a, c)\) is a ground instance of the clause (i), \(R_0(a, b)\) is a negative example of \(R_0\) and \(\text{father}(a, c)\) is true in \(T\). For similar reasons, it contains also \(R_1(a, b, d)\). On the other hand, we have:
\[
E_{a1}^+ = \{\{R_1(a, c, c), R_1(a, e, d)\}, \{R_1(a, f, c), R_1(a, f, d)\}, \{R_1(a, g, c), R_1(a, g, d)\}\}.
\]

For instance, the set \(\{R_1(a, c, c), R_1(a, e, d)\} \in E_{a1}^+\) is built from the set \(\{R_0(a, e)\} \in E_{a0}^\rightarrow\); it means that if the clause (ii) is added to the program, a sufficient condition to ensure that \(R_0(a, e)\) is true is that either \(R_1(a, c, c)\) or \(R_1(a, e, d)\) is true. This explains why we introduce the notion of \(\sigma\)-interpretation, although this notion is more general and could be used in other kinds of learning tasks.

Once the clause (ii) has been built, the learned program is completed by adding either a definition for the predicate \(R_0\) or for the predicate \(R_1\). Since \(R_1\) is incompletely specified, it seems natural to first search a definition for \(R_1\), that will enable to precise it. Let us assume that the learned clause is then:
\[
\text{parent}(X, Y) \rightarrow \text{mother}(Z, Y).
\]

Since the program is not complete, a new clause is built:
\[
\text{parent}(X, Y) \rightarrow \text{father}(Z, Y).
\]
The program is now complete; the variable \(X\) does not appear in the body of the clauses defining \(R_1\), we can then simplify to obtain the program
\[
\begin{align*}
\text{parent}(X, Y) & \rightarrow \text{father}(X, Z), R_1(Y, Z).
\end{align*}
\]
\[
\begin{align*}
\text{mother}(Y, X).
\end{align*}
\]
\[
\begin{align*}
\text{father}(Y, X).
\end{align*}
\]

It is important to notice the the invented predicate \(R_1\) represents the concept parent, which was not given in the initial theory.

### 2.2 Induction process
Let us consider the problem of finding a consistent and \(\sigma\)-complete definition for the target predicate \(R_0\), specified by a \(\sigma\)-interpretation \(E_0\) and let us suppose that the learning process has already introduced some new predicates \(R_j, j > 0\) and built a \(\sigma\)-interpretation \(E_j = \langle E_{a, j}^+, E_j^-\rangle\) specifying
\( R_j \), for each \( R_j, j > 0 \). Let us consider now the problem of finding a clause defining a predicate \( R_i \), \( i \geq 0 \).

A literal \( P(\overline{Y}) \), where \( P \) is either a basic predicate or a predicate \( R_j, j \geq 0 \), is selected, according to its relevance for the predicate \( R_i \). Two situations are distinguished:

- The clause

  \[ (*) \quad R_i(\overline{X}) \leftarrow P(\overline{Y}). \]

  covers no negative examples of \( R_i \) (i.e., no elements of \( E^+_i \)). In this case, the clause \((*)\) is added to the program. Moreover, this clause enables to deduce more precise information about \( R_i \) and about predicates depending\(^2\) on \( R_i \); intuitively, each ground instance \( R_i(\overline{c}) \leftarrow P(\overline{d}) \) of \((*)\) such that \( P(\overline{d}) \) is true implies that \( R_i(\overline{c}) \) is true, which can in turn be used in clauses depending on \( R_i \). Therefore, the set \( I^+_i \) (initially empty) of such grounds atoms is computed and this operation is repeated for each predicate depending on \( R_i \);

- The previous clause \((*)\) covers some negative examples of \( R_i \). In this case, instead of adding the clause \((*)\), we add the clause

  \[ (**) \quad R_i(\overline{X}) \leftarrow P(\overline{Y}), R_k(\overline{Z}). \]

  where \( R_k \) is a new predicate and \( \overline{Z} \) is the union of the variables occurring in \( \overline{X} \) and \( \overline{Y} \). The \( \sigma \)-interpretation \( E_k \), specifying the predicate \( R_k \) is then computed.

To sum up, we propose to build the program by iteratively adding either a clause of the form \((*)\) or a clause of the form \((**)*\), which introduces a new predicate; the form of the clause \((*)\) or \((**)\) is determined by the selected literal. For each predicate \( R_i \) \( i \geq 0 \) (the target predicate or an invented one), two kinds of informations are used: on one hand, the \( \sigma \)-interpretation \( E_k \) gives an imprecise but complete specification of the positive and negative examples of \( R_i \); on the other hand, \( I^+_i \) gives a precise but incomplete interpretation of \( R_i \) since it is a set of positive examples which grows during the learning process. We now describe the way \( E_k \) and \( I^+_i \) are computed.

### 2.2.1 Computation of \( I^+_i \)

Initially, \( I^+_i = \emptyset \). Let us consider the case when a clause

\[ (*) \quad R_i(\overline{X}) \leftarrow P(\overline{Y}). \]

is added to the program, which means that the clause covers no negative examples of \( R_i \). If \( P \) is a basic predicate or a predicate \( R_j \) which does not depend on \( R_i \), then for each ground instance \( R_i(\overline{c}) \leftarrow P(\overline{d}) \) of \((*)\) such that \( P(\overline{d}) \) is true in \( T \) or in \( I^+_i \), \( R_i(\overline{c}) \) is added to \( I^+_i \). The set \( I^+_i \) increases as well as the the sets \( I^+_k \) for the predicates \( R_k \) depending on \( R_i \).

More generally, when a clause \((*) \) or \((**)\) is added, we compute the semantics of \( P \cup T \), where \( P \) is the learned program and we update all the sets \( I^+_k \) according to the new ground atoms that are deduced.

Since the learned program is function symbol free, its semantics is a finite set; it can be obtained by computing the least fixed point of the immediate consequence operator \( T_P \) \cite{5}.

---

\(^1\)We describe in Section 3 the way this literal is selected.

\(^2\)A predicate \( P \) depends on a predicate \( Q \) for a program \( P \) if there exists a clause \( P(\overline{X}) \leftarrow \ldots R(\overline{Y}) \ldots \) in \( P \) such that \( R = Q \) or \( R \) depends on \( Q \).
2.2.2 Computation of $E_i$

When the selected literal $P(Y)$ is such that the clause (*) covers some negative examples of $R_i$, we add to the program the clause

$$(**) \quad R_i(X) \leftarrow P(Y), R_k(Z).$$

where $R_k$ is a new predicate.

Since this predicate has to ensure the consistency of the learned program, the negative examples of $R_k$ are built from the ground instances of this clause which cover negative examples of $R_i$: let $\theta$ be a substitution such that $R_i(\tilde{X})\theta \leftarrow P(\tilde{Y})\theta$. is a ground instance of (*), $R_i(\tilde{X})\theta$ is a negative example of $R_i$ and $P(\tilde{Y})\theta$ is true in $T$ (if $P$ is a basic predicate) or in $I^+_{j}$ (if $P$ is a predicate $R_j$, $j \geq 0$). To ensure $R_i(\tilde{X})\theta$ to be false in the learned program, $R_k(\tilde{Z})\theta$ must be expected to be false. It must therefore be a negative example of the invented literal $R_k$. We obtain this way the set of negative examples of $R_k$:

$$E^-_k = \{ R_k(\tilde{Z}) \mid R_i(\tilde{c}) \leftarrow P(\tilde{d}), R_k(\tilde{d}). \text{ is a ground instance of (**),}
R_i(\tilde{c}) \in E^-_i \text{ and } P(\tilde{d}) \text{ is true in } T \text{ (if } P \text{ is a basic predicate) or in } I^+_{j} \text{ (if } P = R_j) \}$$

Let us notice that when $P$ is an invented predicate $R_j$, more information about $R_j$ can be deduced later by adding new clauses. Therefore, $I^+_j$ can increase. It explains why in SPILP, we check the consistency of the program each time a new clause is added.

The predicate $R_k$ must enable to achieve the $\sigma$-completeness of the learned program $P$, concerning the predicate $R_i$. This enables to define the set $E^{+}_{\sigma_{R_k}}$, defining the positive examples of $R_k$, based on the following idea: let $\sigma \in E^{+}_{\sigma_{R_k}}$, an element of $\sigma$ will be true in $P$, if there exists a ground substitution $\theta$ such that $R_i(\tilde{X})\theta \in \sigma$, $P(\tilde{Y})\theta$ is true in $T$ (if $P$ is a basic predicate) or in $I^+_{j}$ (if $P = R_j$) and $R_k(\tilde{Z})\theta$ is true in $P$. If $E^{+}_{\sigma_{R_k}} = \{\sigma_1, \ldots, \sigma_n\}$, the set $E^{+}_{\sigma_{R_k}}$ is then defined by $\{\sigma'_1, \ldots, \sigma'_n\}$ where

$$\sigma'_p = \{ R_k(\tilde{Z}) \mid R_i(\tilde{c}) \leftarrow P(\tilde{d}), R_k(\tilde{d}). \text{ is a ground instance of (**),}
R_i(\tilde{c}) \in \sigma_p \text{ and } P(\tilde{d}) \text{ is true in } T \text{ (if } P \text{ is a basic predicate)} \text{ or in } I^+_{j} \text{ (if } P = R_j) \}\$$

Nevertheless, let us notice that the learned program has to be $\sigma$-complete w.r.t. $E_0$. It may exist a substitution $\sigma \in E^{+}_{\sigma_{R_k}}, k > 0$, such that no elements of $\sigma$ is true for $P$. This means that we have hypothesized a positive example for $R_k$, incompletely defined by $\sigma$, which appears to be irrelevant for the learning process. It could also happen, even if we have not yet encountered this case, that for all substitutions $\sigma \in E^{+}_{\sigma_{R_k}}, k > 0$, no elements of $\sigma$ is true for $P$. This means that the introduced predicate $R_k$ is irrelevant.

3 The system SPILP

We present now the system SPILP relying on the notions introduced in Section 2. The algorithm adds iteratively either a clause of the form:

$$(*) \quad R_i(X) \leftarrow P(Y).$$
or a clause of the form:

\[(**)
R_i(\overline{X}) \leftarrow P(\overline{Y}), R_k(\overline{Z}).\]

The most important steps in SPILP are the choice of the next predicate \(R_i\) to define and the selection of the literal \(P(\overline{Y})\). Once \(P(\overline{Y})\) is chosen, the form of the added clause is automatically deduced: if no negative example of \(R_i\) is covered by the clause \((*)\), then this clause is added to the program under construction, otherwise the clause \((**)\) is added. In this last case, the \(\sigma\)-interpretation of \(R_k\) is then computed.

In order to choose the next predicate \(R_i\) to learn, we use two heuristics. First, we add a clause for each predicate \(R_i\), for which there exists no clause defining it, i.e., having as head \(R_i(\overline{X})\). This leads to the building of series of clauses, like:

\[
\begin{align*}
R_i(\overline{X}_0) & \leftarrow P_0(\overline{Y}_0), R_{i+1}(\overline{Z}_0), \\
R_{i+1}(\overline{X}_1) & \leftarrow P_1(\overline{Y}_1), R_{i+2}(\overline{Z}_1), \\
& \quad \vdots \\
R_{i+j+1}(\overline{X}_{j+1}) & \leftarrow P_{j+1}(\overline{Y}_{j+1}).
\end{align*}
\]

Such a series is called an **informative definition for \(R_i\)**.

The second heuristic is used to choose the next predicate \(R_i\) to learn, when all the predicates have an informative definition and the learned program is not yet \(\sigma\)-complete. It will be discussed in Section 3.1.

\[
\begin{array}{|l|}
\hline
\mathcal{P} \leftarrow \emptyset \\
\text{While } \mathcal{P} \text{ is not } \sigma\text{-complete} \\
\quad (\exists \sigma \in E_i^+ \text{ s.t. } \sigma \cap I_0^+ = \emptyset) \text{ do} \\
\quad \quad \text{choose a predicate } R_i \text{ to define} \\
\quad \quad \quad \text{build an informative definition for } R_i \\
\quad \text{done} \\
\hline
\end{array}
\]

Main loop of SPILP

Once a predicate \(R_i\) has been chosen, the following procedure **informative** adds to \(\mathcal{P}\) an informative definition for \(R_i\), built from its \(\sigma\)-interpretation \(E_i^\sigma\):

\[
\begin{algorithm}
\text{procedure informative}(\mathcal{P}, R_i, E_i) \\
\quad \text{begin} \\
\quad \quad \text{select a literal } P(\overline{Y}) \\
\quad \quad \text{if } R_i(\overline{X}) \leftarrow P(\overline{Y}) \text{ covers } e \in E_i^- \\
\quad \quad \quad \text{then add } R_i(\overline{X}) \leftarrow P(\overline{Y}), R_k(\overline{Z}) \text{ to } \mathcal{P} \\
\quad \quad \quad \quad \text{(}\(R_j\) is a new predicate) \\
\quad \quad \quad \quad \text{build } E_k \text{ (see Section 2.1.2)} \\
\quad \quad \quad \quad \text{call } \text{informative}(\mathcal{P}, R_k, E_k) \\
\quad \quad \text{else add } R_i(\overline{X}) \leftarrow P(\overline{Y}) \text{ to } \mathcal{P} \\
\quad \quad \quad \quad \text{compute } I_j^\pm \text{ for each } R_j \text{ depending on } R_i \\
\quad \quad \quad \quad \quad \text{(see Section 2.1.1)} \\
\quad \text{end}
\end{algorithm}
\]

Construction of an informative definition for \(R_i\)
SPILP can back-track (on the choice of the predicate \(R_i\) and on the selection of the literal \(P(\bar{Y})\)) before adding the clause \(R_i(\bar{X}) \leftarrow P(\bar{Y})\), all the sets \(I_j^+\) are updated; if a \(I_k^+\) is not consistent w.r.t. \(E_k\) or if the program does not allow to deduce enough informations, the algorithm back-tracks on another literal \(P(\bar{Y})\).

### 3.1 Selection of the literal \(P(\bar{Y})\)

As most systems in ILP, SPILP is based on the notion of coverage to guide the search: the selected literal is the one which covers as many positive examples as possible and which covers as few negative examples as possible. Nevertheless, in our framework, two kinds of information are available for each predicate \(R_i\), \(i \geq 0\): the \(\sigma\)-interpretation \(E_i\) and the set \(I_i^+\) of atoms true for the learned program. Moreover, our description of the positive examples of \(R_i\) is given by a set \(E_{\sigma,j}^+\) containing sets \(\sigma_j\) of ground atoms, each \(\sigma_j\) containing at least a positive example. The program has to be \(\sigma\)-complete only w.r.t. \(E_{\sigma,j}^+\). We define two notions of covering, adapted to our framework, based either on true informations of \(I_i^+\) or on expected information of \(E_{\sigma,j}^+\):

**Definition** Let \(e\) be a ground atom and \(P\) be the program under construction;

- \(e\) is **I-covered by the clause**
  
  \[ C = R_i(\bar{X}) \leftarrow P(\bar{Y}) \]

  iff there exists a ground instance \(C\theta\) of \(C\) such that \(e = R_i(\bar{X})\theta\) and \(P(\bar{Y})\theta\) is true in \(T\) (if \(P\) is a basic predicate) or in \(I_j^+\) (if \(P = R_j\)).

- \(e\) is **I-covered by the clause**
  
  \[ C' = R_i(\bar{X}) \leftarrow P(\bar{Y}), R_k(\bar{Z}) \]

  iff there exists a ground instance \(C'\theta\) of \(C'\) such that \(e = R_i(\bar{X})\theta\), \(P(\bar{Y})\theta\) is true in \(T\) (if \(P\) is a basic predicate) or in \(I_j^+\) (if \(P = R_j\)) and \(R_k(\bar{Z})\theta\) is true in \(I_k^+\).

Notice that the notion of I-covering depends only on information that are deduced from the program \(P \cup T\). Therefore, we have the following result: an example which is I-covered is true for \(P \cup T\).

**Definition** Let \(e\) be a ground atom and let \(P\) be the program under construction;

- \(e\) is **E-covered by the clause**
  
  \[ C = R_i(\bar{X}) \leftarrow P(\bar{Y}) \]

  iff \(e\) is not I-covered by \(C\) and there exists a ground instance \(C\theta\) of \(C\) such that \(e = R_i(\bar{X})\theta\) and \(P(\bar{Y})\theta\) belongs to an element of \(E_{\sigma,j}^+\) (\(P\) is the predicate \(R_j\)).

- \(e\) is **E-covered by the clause**
  
  \[ C' = R_i(\bar{X}) \leftarrow P(\bar{Y}), R_k(\bar{Z}) \]

  iff \(e\) is not I-covered by \(C'\) and there exists a ground instance \(C'\theta\) of \(C'\) such that \(e = R_i(\bar{X})\theta\), \(P(\bar{Y})\theta\) is true in \(T\) (if \(P\) is a basic predicate) or belongs to an element of \(E_{\sigma,j}^+\) (if \(P = R_j\)) and \(R_k(\bar{Z})\theta\) belongs to an element of \(E_{\sigma,k}^+\) or is true in \(I_k^+\).

When an atom \(e\) is E-covered by a clause of \(P\), it is not sure that it will be true for \(P \cup T\).
**Definition**  The notions of I-covering and E-covering are extended to sets of ground atoms: a set $\sigma$ of ground atoms is I-covered (resp. E-covered) iff there exists at least an atom of $\sigma$ I-covered (resp. E-covered).

The literal $P(Y)$ that is chosen for the definition of $R_i(X)$ is the one which has the largest information gain among all possible literals. We have adapted the information gain used in FOIL [14] in our framework, in order to take into account not only E-covered atoms, but also I-covered ones.

Let $E_{i,\sigma}^+ = \{\sigma_1, \ldots, \sigma_n\}$ and $\Sigma_i = \sigma_1 \cup \sigma_2 \cup \ldots \cup \sigma_n$. Let $C$ be the clause $R_i(X) \leftarrow P(Y)$. We define

- $n_I^+$: the number of ground atoms in $\Sigma_i - I_i^+$,
- $n_I^-$: the number of ground atoms of $E_i^+$,
- $\hat{n}_I^+$: the number of ground atoms of $\Sigma_i - I_i^+$ which are I-covered by $C$,
- $\hat{n}_I^-$: the number of ground atoms of $E_i^+$ which are I-covered by $C$,
- $\hat{c}_I^+$: the number of non-empty $\sigma_i - I_i^+$ I-covered by $C$, $\sigma_i \in E_{i,\sigma}^+$.

The information gain of $P(Y)$ w.r.t the notion of I-covering is defined by

$$\text{gain}_I(P(Y)) = \hat{c}_I^+ \times (\log_2(\frac{\hat{n}_I^+ + \hat{n}_I^-}{\hat{n}_I^+ + \hat{n}_I^-}) - \log_2(\frac{n_I^+}{n_I^+ + n_I^-}))$$

We define in the same way

- $\check{n}_E^+$: the number of ground atoms of $\Sigma_i - I_i^+$ which are E-covered by $C$,
- $\check{n}_E^-$: the number of ground atoms of $E_i^+$ which are E-covered by $C$,
- $\check{c}_E^+$: the number of non-empty $\sigma_i - I_i^+$ E-covered by $C$, $\sigma_i \in E_{i,\sigma}^+$.

The information gain of $P(Y)$ w.r.t the notion of E-covering is defined by

$$\text{gain}_E(P(Y)) = \check{c}_E^+ \times (\log_2(\frac{\check{n}_E^+ + \check{n}_E^-}{\check{n}_E^+ + \check{n}_E^-}) - \log_2(\frac{n_E^+}{n_E^+ + n_E^-}))$$

The total information gain is defined by:

$$\text{gain}(P(Y)) = \omega_I^* \text{gain}_I(P(Y)) + \omega_E^* \text{gain}_E(P(Y))$$

where $\omega_I$ and $\omega_E$ are such that $\omega_I + \omega_E = 1$. A similar idea was developed in [15], although it was in a different context. These coefficients allow to control the influence of the information contained in the $\sigma$-interpretations to select the literal: the more $\omega_I$ is high, the more SPIILP uses information deduced from the previous learned clauses; it is better to choose a high $\omega_I$ when the target predicate is defined with an incomplete specification. On the other hand, the more $\omega_E$ is high, the more the selected literal will build clauses which do not enable to deduce new information and the algorithm might back-track more. Practically, we use the values $\omega_I = 0.8$ and $\omega_E = 0.2$. 

9
3.2 Choice of the predicate $R_i$

Once an informative definition has been built for the target predicate $R_0$, SPILP has to choose a predicate for which a new informative definition is built and this choice must be repeated until the program is $\sigma$-complete w.r.t $E_0$.

To determine the next predicate $R_i$, the total information gain in the clause

$$R_j(\bar{X}) \leftarrow P_{jk}(\bar{Y}).$$

is computed for each possible $R_j$, $j \geq 0$ and for each possible literal $P_{jk}(\bar{Y})$. This gives the total information gain for a set of pairs $(R_j(\bar{X}), P_{jk}(\bar{Y}))$; the predicate $R_i$ that is chosen is the one that maximizes the gain of the pairs $(R_j(\bar{X}), P_{jk}(\bar{Y}))$. When a large number of predicates has been invented, this choice is time-consuming but it gives both the predicate and the selected literal ($P(\bar{Y})$). As has already been mentioned, the clause that must be added is then automatically determined depending on whether negative examples are covered.

3.2.1 A particular case

In some cases, the pair having the best gain has the form $(R_i(\bar{X}), R_i(\bar{Y}))$. It may happen when an informative definition for $R_i$ already exists and information has already been deduced for $R_i$. If we add the clause:

(i) $R_i(\bar{X}) \leftarrow R_i(\bar{Y}), \ldots$

to the program, and if another clause defining $R_i$ had been previously added, the clause (i) may allow to build new negative examples of $R_i$, which can make previous clauses inconsistent. For this reason, we introduce a new predicate $R_k$: if the clauses defining $R_i$ are

$$R_i(\bar{X}) \leftarrow B_1,$$

$$\ldots$$

$$R_i(\bar{X}) \leftarrow B_n,$$

these clauses are replaced by the clauses

$$R_k(\bar{X}) \leftarrow B_1,$$

$$\ldots$$

$$R_k(\bar{X}) \leftarrow B_n,\quad R_i(\bar{X}) \leftarrow R_k(\bar{X}),$$

and the pair $(R_i(\bar{X}), R_k(\bar{Y}))$ is then considered to be the pair that maximizes the gain.

An illustration of this case is given in Section 4, when learning the predicate ancestor.

3.3 Biases-Simplifications

A first bias has been mentioned above: it consists in choosing values for the parameters $\omega_I$ and $\omega_E$. The selection of the literal $P(\bar{Y})$ is also biased when several literals have the same information gain: SPILP chooses first basic predicates in order to increase the simplicity of the learned program and in order to complete the definition of non-basic predicates. Finally, SPILP uses two biases which limit the number of new variables: when the clause

$$R_i(\bar{X}) \leftarrow P(\bar{Y}), R_k(\bar{Z}).$$

is built, the variables in $\bar{Z}$ is the union of the variables appearing in $\bar{X}$ and $\bar{Y}$. Each new variable introduced in $\bar{Y}$ increases the arity of $R_k$; moreover, the complexity of the selection of a literal $P'(\bar{Y})$ for the definition of $R_k(\bar{Z})$ depends mainly on the arity of $R_k$ since SPILP, as FOIL, explores the space of all possible predicates and all possible variabilities. For this reasons, SPILP uses the 2 following biases:
• the number of new variables in a variabilization of a predicate \( R_i \) is limited to 2,

• the information gain is divided by \((2 + \text{New})\), where \text{New} is the number of variables introduced by the literal.

To conclude this section, let us consider the way the learned programs can be simplified, even though it is not yet implemented. The first simplification concerns the invented predicates which are defined by just one clause: if the clause is recursive, it is deleted otherwise it is removed by unfolding. For the remaining invented predicates, the simplification concerns the arity of the predicate: if a variable appears in the head of the definition of a \( R_i \) and if it does not appear in the body of any clauses defining \( R_i \), this variable is removed, which reduces the arity of \( R_i \). This operation is repeated for invented predicates depending on \( R_i \). In next sections, we present some programs learned by SPLIP.

4 Examples

The algorithm SPLIP has been written in Prolog, the following programs have been built in 5 up to 20 seconds on a Sparc 4. We give first the programs built by SPLIP and then we rewrite them using simplifications given in Section 3.3 and renaming the invented predicates. The 3 following programs have been learned on a family containing 11 people \( \{a, b, \ldots, k\} \); the basic predicates are \textit{father} and \textit{mother}, \textit{parent} is a basic predicate only in the third example. The initial theory \( T \) is defined by:

\[
\{ \text{father}(a, c), \text{father}(a, d), \text{father}(c, e), \text{father}(g, j), \\
\text{father}(g, i), \text{father}(h, k), \text{mother}(b, c), \text{mother}(b, d), \\
\text{mother}(d, f), \text{mother}(d, g), \text{mother}(e, h) \}\]

4.1 Learning grand-father

The first example is the one proposed in Section 2; the target predicate is \textit{grand-father} (gf) and the predicate \textit{parent} is not given in \( T \). The program learned by SPLIP is given in fig. 4.1a. As mentioned in Section 3, \( R_1 \) can be simplified and renamed with \textit{parent}; we get the program of fig 4.1b (\textit{parent}(X, Y) is true here if \( Y \) is a parent of \( X \)).

\[
\begin{align*}
\text{R0}(X, Y) &< - \text{father}(X, Z) \land \text{R1}(X, Y, Z). \\
\text{R1}(X, Y, Z) &< \text{mother}(Z, Y). \\
\end{align*}
\]

\[
\begin{align*}
gf(X, Y) &< - \text{father}(X, Z), \text{parent}(Y, Z). \\
\text{parent}(X, Y) &< \text{mother}(Y, X). \\
\end{align*}
\]

fig. 4.1a  \hspace{1cm} fig. 4.1b

4.2 Learning ancestor

This example illustrates the particular case of predicate invention discussed in Section 3.2: SPLIP has to learn \textit{ancestor} (\textit{anc}) with the basic predicates \textit{father} and \textit{mother}. It first builds the 2 clauses:

\[
\begin{align*}
\text{R0}(X, Y) &< - \text{father}(X, Y). \\
\text{R0}(X, Y) &< - \text{mother}(X, Y). \\
\end{align*}
\]

When SPLIP tries to build a new clause defining \( R_0(X, Y) \), the selected literal is \( R_0(X, Z) \) which can easily be explained since the system mainly uses information of \( L_0^+ \) to make its choice. We can
notice, although SPILP does not know this, that at this level information known about \( R_0 \) are information defining \textit{parent}. Since the system cannot build the clause \( R_0(X, Y) \leftarrow R_0(X, Z), \ldots \), it renames the learned concept \( R_0 \) in \( R_1 \) and finally, we get the program of fig 4.2a. \( R_2 \) is defined by just one clause so it can be deleted by unfolding techniques; then if \( R_1 \) is renamed with \textit{parent}, we get the program of fig 4.2b.

\[
\begin{align*}
R_0(X,Y) & \leftarrow R_1(X,Y). \\
R_0(X,Y) & \leftarrow R_1(X,Z), R_2(X,Y,Z). \\
R_1(X,Y) & \leftarrow \text{father}(X,Y). \\
R_1(X,Y) & \leftarrow \text{mother}(X,Y). \\
R_2(X,Y,Z) & \leftarrow R_0(Z,Y).
\end{align*}
\]

fig. 4.2a

\[
\begin{align*}
\text{anc}(X,Y) & \leftarrow \text{parent}(X,Y). \\
\text{anc}(X,Y) & \leftarrow \text{parent}(X,Z), \text{anc}(Y,Z). \\
\text{parent}(X,Y) & \leftarrow \text{mother}(X,Y). \\
\text{parent}(X,Y) & \leftarrow \text{father}(X,Y).
\end{align*}
\]

fig. 4.2b

### 4.3 Learning male-ancestor

The goal here is to find a definition for \textit{male - ancestor (ma)} with the basic predicates \textit{father, mother} and \textit{parent}. In this problem, the invention of a new predicate such as \textit{female - ancestor} or \textit{ancestor} is essential. The learned program is given in fig. 4.3a. In this program, \( R_2 \) can be removed by unfolding, then the variable \( X \) does not appear in the body of the definitions of \( R_1 \) which is simplified and renamed with \textit{ancestor (anc)}. We obtain the program of fig. 4.3b (\( \text{anc}(Y,Z) \) is true if \( Z \) is an ancestor of \( Y \)):

\[
\begin{align*}
R_0(X,Y) & \leftarrow \text{father}(X,Y). \\
R_0(X,Y) & \leftarrow \text{father}(X,Z), R_1(X,Y,Z). \\
R_1(X,Y,Z) & \leftarrow \text{parent}(Z,Y). \\
R_1(X,Y,Z) & \leftarrow \text{parent}(Z,T), R_2(X,Y,Z,T). \\
R_2(X,Y,Z,T) & \leftarrow R_1(Z,Y,T).
\end{align*}
\]

fig. 4.3a

\[
\begin{align*}
\text{ma}(X,Y) & \leftarrow \text{father}(X,Y). \\
\text{ma}(X,Y) & \leftarrow \text{father}(X,Z), \text{anc}(Y,Z). \\
\text{anc}(Y,Z) & \leftarrow \text{parent}(Z,Y). \\
\text{anc}(Y,Z) & \leftarrow \text{parent}(Z,T), \text{anc}(Y,T).
\end{align*}
\]

fig. 4.3b

### 5 Complexity - Further works

We propose in this paper a new system, SPILP, for constructive learning predicate definitions. This system performs empirical learning of function free logic programs and it is able to invent both predicates simplifying the learned program and predicates defined recursively. Moreover, the framework that we have developed enable to learn when the specification of the target predicate is both incomplete and imprecise, since SPILP relies both on information from the specification and on information deduced from the learned clauses.

SPILP can be compared with FOIL algorithm [14]: if the chosen predicate is always \( R_0 \) and if the selected literals are restricted to basic predicates, SPILP works in a way similar to FOIL. Nevertheless, three main differences can be noted, namely the ability to deal with imprecise information, the computation of the gain and the fact that SPILP always builds a consistent and complete program whereas FOIL produces a program which covers all the positive examples and no negative ones.

Since when building a clause, FOIL and SPILP explore the space of all possible literals, we can compare the complexity of SPILP with the one of FOIL which is known to be a fast algorithm; at the beginning of learning, the complexity is the same, but once a predicate is invented by SPILP, the search space of all the possible literals is widen by new predicates. The bias which limits the
number of introduced variables in the variabilization of an invented predicate limits this search space to a reasonable size.

Future works include:

- handling negation by building clauses of the form: \( R_k(\bar{X}) \leftarrow P(\bar{Y}), \neg R_k(\bar{Z}). \)

- extending the form of the learned clauses: once an informative definition is built for the target predicate, the clause
  \[ R_0(\bar{X}) \leftarrow P(\bar{Y}), R_1(\bar{Z}). \]
  could be transformed into
  \[ R_0(\bar{X}) \leftarrow R_k(\bar{Y}), R_1(\bar{Z}). \]
  \[ R_k(\bar{X}) \leftarrow P(\bar{Y}). \]
  where \( R_k \) is a new predicate. It could then be possible to learn a concept defined as the conjunction of several concepts, which are defined recursively.

References


