Learning definitions in Answer Set Programming

Ravi Palla

1. Introduction

Inductive Logic Programming (ILP) (Muggleton, 1990) is a branch of Machine Learning that deals with learning theories in the form of logic programs. The current approaches in ILP mainly deal with learning classical clausal theories, especially Horn Logic Programs. However, it is well known that classical Horn Logic programs are not expressive enough to represent and reason about various domains involving incomplete information. We need to employ non-monotonic reasoning to effectively reason about domains with incomplete information. The Stable Model semantics introduced in (Gelfond & Lifschitz, 1988) provides a powerful framework for non-monotonic reasoning. Answer Set Programming (ASP) is a declarative programming paradigm useful for solving combinatorial search problems. Definitions (Erdogan & Lifschitz, 2003) in ASP are used to represent various non-elementary concepts. For example, the following rules define path:

\[
\begin{align*}
\text{path}(x,y) & \leftarrow \text{edge}(x,y). \\
\text{path}(x,y) & \leftarrow \text{path}(x,z), \text{path}(z,y).
\end{align*}
\]

(Sakama, 2001) presents an approach for learning in ASP. Since ASP is based on the Stable Model semantics, the algorithms described in (Sakama, 2001) and (Sakama, 2005) enable learning of non-monotonic theories. Given background knowledge in the form of an answer set program and sets of positive and negative examples, the papers present an approach that enables learning of new definitions such that the resulting program entails all the positive examples and does not entail any negative example. In this paper, we extend the work presented in those papers. For this work, we will consider logic programs extended with classical negation (Gelfond & Lifschitz, 1991) inline with (Sakama, 2005).

We begin by briefly discussing answer set programs extended with classical negation (also called as extended logic programs) and the approach in (Sakama, 2005). We then recognize some of the problems with the approach and discuss ways to avoid some of them. We follow it up by providing soundness and completeness results in view of the improvements presented in this paper. We then discuss about the monotonicity property and the computational complexity of the learning approach before concluding.

2. Extended Logic Programs

An extended logic program \( \Pi \) is a set of rules of the form

\[
L_0 \leftarrow L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n.
\]

(3)

where \( n \geq m \geq 0 \) and each \( L_i \) is a literal. Let \( \text{Lit} \) be the set of all ground literals in the language of \( \Pi \). For any rule (3), \( L_0 \) is the head of the rule and \( L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n \) is the body of the rule. \( L_1, \ldots, L_m \) is the positive body and \( \text{not } L_{m+1}, \ldots, \text{not } L_n \) is the negative body. If the head is empty, then the rule is a constraint and if the body is empty, the rule is a fact. We will first define the notion of answer sets for a program without variables. For any set \( S \subseteq \text{Lit} \), let \( \Pi^S \) be the program obtained from \( \Pi \) by deleting

- all rules with \( \text{not } L \) in the body, where \( L \in S \), and
- all \( \text{not } L \) from the bodies of the remaining rules.

It is clear that \( \Pi^S \) does not have any rules with default negation \( \text{not} \). \( S \) is an answer set of \( \Pi \) iff \( S \) is the smallest set such that

- for any rule

\[
L_0 \leftarrow L_1, \ldots, L_m
\]

in \( \Pi^S \), \( L_0 \in S \) if \( L_1, \ldots, L_m \in S \), and

- \( S \) does not contain a pair of complimentary literals \( L \) and \( \lnot L \).

For example, consider the following program \( \Pi \):

\[
\begin{align*}
p & \leftarrow \lnot q, \text{not } r. \\
q. \\
r & \leftarrow \text{not } p.
\end{align*}
\]
Consider the set $S = \{ \neg q, p \}$. Then $\Pi^S$ is

\[
\begin{align*}
p & \leftarrow \neg q. \\
\neg q & .
\end{align*}
\]

To check if $S$ is an answer set, we need to check if $S$ is the smallest set such that $\neg q \in S$ (since the body of the second rule of $\Pi^S$ is empty) and $p \in S$ (since $\neg q \in S$). Since this is the case, $S$ is an answer set of $\Pi$. The Herbrand universe of any program $\Pi$ (possibly containing variables) is the set of all constants that can be constructed from the object and function constants in the language of $\Pi$. To compute the answer sets of a program with variables, we compute the answer sets of the variable-free program obtained by substituting the variables in the program with the constants in the Herbrand universe in every possible way. We say that $\Pi \models L$ if every answer set of $\Pi$ contains the literal $L$.

3. Learning from positive examples

Given background knowledge in the form of an extended logic program $P$ and a ground literal $L$ representing a positive example such that $P \not\models L$, the goal here is to learn a rule $R$ such that $P \cup R$ is consistent (i.e., has an answer set) and $P \cup R \models L$. It is assumed that $P,R$ and $L$ share the same language. We will first need to define some concepts before proceeding to the actual approach defined in (Sakama, 2005).

$\text{Lit}$ denotes the set of all ground literals in the language of $P$. $\text{LP-literal}$ denotes the set $\text{Lit} \cup \{ \text{not } L \mid L \in \text{Lit} \}$. Given any set $S \subseteq \text{Lit}$, $S^+ = S \cup \{ \text{not } L \mid L \in \text{Lit} \setminus S \}$ denotes the expansion set of $S$. For any LP-literal $L$, $\text{pred}(L)$ denotes the predicate associated with $L$ and $\text{const}(L)$ denotes the set of constants occurring in $L$ (not including the predicate constant). For any LP-literal $L$, $|L|$ denotes $L$ if $L$ is a literal and $L_1$ if $L$ is of the form $\text{not } L_1$.

$\text{LP-Literal } L_1$ is relevant to $\text{LP-literal } L_2$ if

- $\text{pred}(L_1) = \text{pred}(L_2)$ and $\text{const}(L_1) = \text{const}(L_2)$ or
- for some $\text{LP-literal } L_3$, $\text{pred}(L_1) \cap \text{const}(L_3) \neq \emptyset$ and $L_3$ is relevant to $L_2$.

An LP-literal $L$ is involved in a program $P$ if $|L|$ occurs in the ground instance of $P$.

For example, consider the following program $P$:

\[
\begin{align*}
p(x) & \leftarrow q(x). \\
q(x) & \leftarrow \text{not } r(x,y). \\
s(x) & \leftarrow \neg r(x,b). \\
p(a) & .
\end{align*}
\]

In $P$, $p(a)$ is relevant to $p(a)$, $q(a)$, $\neg q(a)$ and $r(a,b)$; $q(a)$ shares a constant with $r(a,b)$ and $r(a,b)$ is relevant to $q(b)$; $\neg r(a,b)$ is relevant to $r(a,b)$, $r(b,a)$, $q(b)$ and $\neg p(a)$. $p(a)$ and $\neg p(a)$ are involved in $P$ where $\neg p(a)$ and $\neg \neg p(a)$ are not. $\neg r(a,b)$ is involved in $P$ where $\neg r(b,a)$ is not.

We are now ready to present the approach to learning from positive examples as defined in (Sakama, 2005).

Given background knowledge in the form of a function-free extended logic program $P$ that has exactly one answer set, and a ground literal $L$ as a positive example, the procedure (algorithm) to learn the rule $R$ such that $P \cup R \models L$ is as follows:

- Compute the answer set $S$ of $P$.
- Compute the expansion set $S^+$ of $S$.
- Construct a constraint $\leftarrow \Gamma$ where $\Gamma$ consists of all the elements in the set $\{ L_1 \mid L_1 \in S^+, L_1$ is relevant to $L$ and is involved in $P \cup L \}$. Since $P \not\models L$, $not L \in S^+$.
- Form the ground rule $L \leftarrow \Gamma'$ from $\leftarrow \Gamma$ by moving $not L$ to the head as $L$.
- $R$ is any rule such that for some substitution $\theta$, $R\theta = L \leftarrow \Gamma'$.

Example 1: Consider the following background program $P$:

\[
\begin{align*}
bird(x) & \leftarrow penguin(x). \\
bird(tweety) & . \\
penguin(polly) & .
\end{align*}
\]

and the positive example $L = \text{flies(tweety)}$.

$P$ has only one answer set $S = \{ \text{bird(tweety)}, \text{bird(polly)}, \text{penguin(polly)} \}$. So, $P \not\models L$.

$S^+ = \{ \text{bird(tweety)}, \text{bird(polly)}, \text{penguin(polly)}, \text{not } \neg \text{bird(tweety)}, \text{not } \neg \text{bird(polly)}, \text{not } \neg \text{penguin(polly)}, \text{not } \neg \text{penguin(tweety)}, \text{not } \neg \text{penguin(tweety)}, \text{not } \text{flies(tweety)}, \text{not } \neg \text{flies(tweety)}, \text{not } \text{flies(polly)}, \text{not } \neg \text{flies(polly)} \}$.

Picking all the LP-literals from $S^+$ that are relevant to $L$ and are involved in $P \cup L$, we get the constraint

\[
\leftarrow \text{bird(tweety)}, \text{not } \text{penguin(tweety)}, \text{not } \text{flies(tweety)}.
\]
Moving not \textit{flies(tweety)} to the head as \textit{flies(tweety)}, we get
\[
\text{flies(tweety)} \leftarrow \text{bird(tweety)}, \neg \text{penguin(tweety)}.
\]
(7)

Considering the substitution \(\theta = x/tweety\), we get \(R\) as
\[
\text{flies}(x) \leftarrow \text{bird}(x), \neg \text{penguin}(x).
\]
(8)

If \(P\) has multiple answer sets, a rule \(R\) is learnt with respect to each answer set \(S\) of \(P\). If we are given a set of positive examples \(E\) instead of just a literal \(L\), we apply the same learning approach for each literal \(L \in E\), considering the same background program \(P\) for all \(L \in E\).

Example 2: Given the same program \(P\) as in example 1 and the set of positive examples \(E = \{\text{flies(tweety)}, \neg \text{flies(polly)}\}\),

\(R_1: \text{flies}(x) \leftarrow \text{bird}(x), \neg \text{penguin}(x)\)

is learnt from the positive example \(L = \text{flies(tweety)}\) and

\(R_2: \neg \text{flies}(x) \leftarrow \text{bird}(x), \text{penguin}(x)\)

is learnt from the positive example \(L = \neg \text{flies(polly)}\).

In the following subsections, we will discuss the problems with this approach.

### 3.1. Premature Generalization

The last step in the learning approach defined in section 3 defines the learnt rule \(R\) as any rule such that for some substitution \(\theta\), \(R\theta = L \leftarrow \Gamma\). This step is nothing but generalizing the rule \(L \leftarrow \Gamma\) by a process that can be considered as inverse-substitution. Such basic generalization leads to problems in certain cases.

Example 3: Consider the following background knowledge \(P\)

\[
\begin{align*}
\text{bird(tweety)}. \\
\text{bird(polly)}. \\
\end{align*}
\]

and the set of positive examples \(E = \{\text{flies(tweety)}, \neg \text{flies(polly)}\}\). The two rules learnt are

\[
\begin{align*}
\text{flies}(x) & \leftarrow \text{bird}(x). \quad (9) \\
\neg \text{flies}(x) & \leftarrow \text{bird}(x). \quad (10)
\end{align*}
\]

Clearly (9) and (10) are contradictory which implies \(P \cup (9) \cup (10)\) is inconsistent. (9) is a generalization of
\[
\text{flies(tweety)} \leftarrow \text{bird(tweety)}. \quad (11)
\]

and (10) is a generalization of
\[
\neg \text{flies(polly)} \leftarrow \text{bird(polly)}. \quad (12)
\]

The inconsistency here is as a result of what can be called as premature generalization. In such cases, the inconsistency can be avoided by considering the ground rules \(L \leftarrow \Gamma\) as the rules learnt until further learning justifies the removal of these rules. Considering the set of such ground rules learnt to be \(Q\), any set of rules \(R \subseteq Q\) can be removed if at any stage, \((P' \setminus R) \models \text{head}(R_1)\) for all \(R_1 \in R\), where \(P' \supset P\) is the knowledge consisting of all the learnt rules upto that stage. In other words, if a positive example \(\text{head}(R_1)\) is entailed by the program obtained by deleting \(R_1\), then \(R_1\) is redundant and can be removed.

Consider another example where in the background knowledge \(P\) is empty and the positive example \(L = \text{penguin(polly)}\). The rule learnt in this case is
\[
\text{penguin}(x). \quad (13)
\]

indicating that all elements in the domain are penguins which might not be true. This problem can also be avoided using the above approach in which case the rule learnt is the fact \(\text{penguin(polly)}\).

A lot of examples fall into the premature generalization category. For example, given the background knowledge \(P = \text{bird(tweety)}\) and the positive example \(L = \text{flies(tweety)}\). The rule learnt is
\[
\text{flies}(x) \leftarrow \text{bird}(x). \quad (14)
\]

Since not all birds fly, this rule can also be considered as a rule obtained as a result of premature generalization. However, it is not possible to automatically identify all the rules resulting from such generalizations. In this subsection, we have defined two conditions (inconsistency and empty body) under which premature generalization is (almost certainly) done during rule learning.

### 3.2. No Level based concept learning

Example 4:

Consider the following background knowledge \(P\):

\[
\begin{align*}
\text{bird(tweety)}. \\
\text{bird(polly)}. \\
\neg \text{crippled(polly)}. \\
\text{nightingale(tweety)}. \\
\end{align*}
\]

and the positive example \(L = \text{sings(tweety)}\). In this case, the following rule is learnt:
\[
\text{sings}(x) \leftarrow \text{bird}(x), \text{nightingale}(x). \quad (15)
\]
Now consider the positive example \( L = \text{walks}(polly) \). The new rule learnt using \( P \cup \{15\} \) as background knowledge is

\[
\text{walks}(x) \leftarrow \text{bird}(x), \neg\text{crippled}(x), \\
\quad \neg\text{sings}(x), \neg\text{nightingale}(x).
\]  

(16)

Clearly, (16) is unintuitive as a bird able to sing is not related to a bird able to walk.

One way to get around this problem is through level based learning. Generally, a concept can have a level associated with it. Concepts at higher levels are defined only in terms of concepts at lower levels. In certain cases, concepts can be defined in terms of themselves (for ex: (2) defines \text{path} in terms of itself).

Level based concept learning can be accomplished by adding a preprocessing step to the learning approach defined in section 3. Given background knowledge \( P \) and a positive example \( L \), the preprocessing step is to remove all the rules from \( P \) that define any concept that is at the same or higher level compared to the concept associated with \( L \). Since a concept can be defined in terms of itself, the rules that define the concept associated with \( L \) are not removed. In example 4, \text{nightingale} and \neg\text{crippled} can be associated with level 1, \text{bird} can be associated with level 2, and \text{walks} and \text{sings} can be associated with level 3. Given the background knowledge \( P \cup \{15\} \) and the positive example \( L = \text{walks}(polly) \), the preprocessing step removes the rule (15) so that the rule learnt changes to

\[
\text{walks}(x) \leftarrow \text{bird}(x), \neg\text{crippled}(x).
\]  

(17)

As shown in the above example, the level based concept learning approach can prevent learning of certain unintuitive rules. However, the drawback of this approach is that concepts need to have levels associated with it. It is thus better to have the association of a concept with a level as an option rather than as a condition. In other words, a concept can have an associated level but it is not mandatory. The preprocessing step is applied only if the positive example \( L \) has a level associated with it, and any rule that defines a concept with no associated level is not deleted in the preprocessing step.

3.3. Learning depends on the order of training examples

The rules learnt using the approach defined in section 3 depend on the order in which the training examples are provided.

Example 5: Consider the domain consisting of whole numbers, the following background knowledge \( P \)

\[
\text{divisibleby2}(0).
\]

\[
\text{divisibleby2}(x + 2) \leftarrow \text{divisibleby2}(x).
\]

and the training example \( E = \{\text{even}(0), \text{odd}(3)\} \). The rules learnt are

\[
\text{even}(x) \leftarrow \text{divisibleby2}(x).
\]

(18)

\[
\text{odd}(x) \leftarrow \neg\text{divisibleby2}(x).
\]

(19)

However, if the training examples are given in the order \( E_1 = \text{even}(0), E_2 = \text{odd}(3) \) the rules learnt are

\[
\text{even}(x) \leftarrow \text{divisibleby2}(x).
\]

\[
\text{odd}(x) \leftarrow \neg\text{divisibleby2}(x), \neg\text{even}(x).
\]  

(20)

(20), though correct, is redundant. Ideally, we would like to learn (18) and (19). This problem can however be avoided in this case by using level based concept learning as explained in section 3.2. Here, we can set \text{odd} and \text{even} to the same level resulting in the learning of the rules (18) and (19).

Now, consider the following example:

Example 6: Consider again the domain consisting of whole numbers, the background knowledge \( P = \emptyset \) and the training examples in the order \( E_1 = \text{odd}(3), E_2 = \text{number}(3) \). The rules learnt are

\[
\text{odd}(x).
\]

\[
\text{number}(x) \leftarrow \text{odd}(x).
\]

Since the body of the first rule is empty, applying the approach to handle premature generalization, the rules learnt change to

\[
\text{odd}(3).
\]  

(21)

\[
\text{number}(x) \leftarrow \text{odd}(x).
\]

(22)

Now, consider the training examples in the order \( E_1 = \text{number}(3), E_2 = \text{odd}(3) \). The rules learnt are

\[
\text{number}(x).
\]

\[
\text{odd}(x) \leftarrow \text{number}(x).
\]

(23)

Here, (23) is unintuitive as it says that all numbers are \text{odd}. Using level based learning, by setting \text{odd} to be of a lower level than \text{number} results in learning of the rules

\[
\text{number}(x).
\]

\[
\text{odd}(x).
\]

(24)
Since the body of the learnt rules is empty, we can assume premature generalization and apply the approach described in section 3.1. In this case, the rules learnt are

\[
\text{number}(3), \\
\text{odd}(3).
\]  

(25)

However, (21) and (22) are still preferable compared to the above rules.

As we saw in example 6, level based learning and the approach to handling premature generalization are not sufficient to handle certain problems arising due to the dependence of the learning on the order of training examples.

4. Learning from negative examples

In this section, we will review the approach to learning from negative examples as defined in (Sakama, 2005), and discuss ways to improve it. We need to define a few concepts before actually explaining the algorithm. The dependency graph of a program \( P \) is a directed graph where the vertices are the predicates occurring in \( P \) and there is a positive edge (resp. negative edge) from predicate \( p_1 \) to a predicate \( p_2 \) if there is a rule such that \( p_1 \) occurs in the head and \( p_2 \) occurs in the positive body (resp. negative body). Predicate \( p_1 \) depends on \( p_2 \) (in \( P \)) if there is a path from \( p_1 \) to \( p_2 \). \( p_1 \) strongly depends on \( p_2 \) if in every path containing \( p_1 \), \( p_1 \) depends on \( p_2 \). \( p_1 \) negatively depends on \( p_2 \) if a path from \( p_1 \) to \( p_2 \) contains odd number of negative edges. The dependencies between literals \( L_1 \) and \( L_2 \) in program \( P \) are defined in the same way. A program \( P \) has a negative cycle if any predicate negatively depends on itself. A rule \( R \) is negative-cycle-free if there is no negative edge from \( \text{pred}(\text{head}(R)) \) to itself.

Consider the following program \( P \):

\[
\begin{align*}
\text{p}(x) & \leftarrow \neg \text{q}(x). \\
\text{q}(x) & \leftarrow \neg \text{r}(x). \\
\text{r}(x) & \leftarrow \text{s}(x).
\end{align*}
\]

The predicate dependency graph of \( P \) contains vertices \( p,q,r \) and \( s \). The graph contains

- a negative edge from \( p \) to \( q \) because of the first rule,
- a negative edge from \( q \) to \( r \) because of the second rule,
- a positive edge from \( q \) to \( s \) because of the third rule, and
- a negative edge from \( r \) to \( r \) because of the last rule.

As a result, \( p \) strongly and negatively depends on \( q \); \( p \) negatively depends on \( s \) but does not depend strongly on \( s \) because there is a path \((p, q)\) that does not contain \( s \). The first 3 rules are negative-cycle-free but the last rule is not since there is a negative edge from \( r \) to itself.

Extending the same concept to literals, for any constants \( a \) and \( b \), \( p(a) \) strongly and negatively depends on \( q(a) \) but does not depend on \( q(b) \). Similarly, \( p(a) \) depends negatively on \( s(a) \) but does not depend on \( s(b) \).

Given background knowledge in the form of a function-free extended logic program \( P \) that has exactly one answer set, a ground literal \( L \) representing a negative example and a predicate \( \text{pred}(K) \) to learn where \( L \) strongly and negatively depends on literal \( K \) in \( P \), the procedure (algorithm) for learning a rule \( R \) such that \( P \cup R \not\models L \) is as follows (here it is assumed that \( P \models L \)):

- Compute the answer set \( S \) of \( P \).
- Compute the expansion set \( S^+ \) and the constraint \( \leftarrow \Gamma \) from the set \( \Gamma \subseteq S^+ \) made up of all LP-literals that are relevant to \( L \) and are involved in \( P \cup L \).
- Form the rule \( K \leftarrow \Gamma' \) by shifting \( \not\Gamma \) in the body to the head.
- Form the rule \( R \) such that \( R\theta = K \leftarrow \Gamma'' \) for some substitution \( \theta \) where \( \Gamma'' \) is obtained from \( \Gamma' \) by dropping all the LP-literals with a predicate that strongly and negatively depends on \( \text{pred}(K) \) in \( P \).

Example 7: Consider the following background program \( P \) and the negative example \( L = \text{flies}(\text{polly}) \):

\[
\begin{align*}
\text{flies}(x) & \leftarrow \text{bird}(x), \not\text{ab}(x). \\
\text{bird}(x) & \leftarrow \text{penguin}(x). \\
\text{penguin}(\text{polly}).
\end{align*}
\]

(26)

Here \( P \models \text{flies}(\text{polly}) \). The first rule of \( P \) implies that if a bird is not \( \text{abnormal} \), then it flies. The abnormality predicate \( \text{ab} \) here is useful in finding
conditions under which a bird does not fly. Let \( ab \) be the predicate to learn. Note that \( \text{flies} \) negatively and strongly depends on \( ab \) and \( \text{flies}(\text{polly}) \) strongly and negatively depends on \( \text{ab}(\text{polly}) \). The goal is to construct a rule \( R \) such that \( P \cup R \not= \text{flies}(\text{polly}) \). The answer set of \( P \) is 
\[
\{ \text{bird}(\text{tweety}), \text{penguin}(\text{polly}), \text{bird}(\text{polly}), \text{flies}(\text{tweety}), \text{flies}(\text{polly}) \}.
\]
The expansion set \( S^+ \) is therefore 
\[
\{ \text{bird}(\text{tweety}), \text{bird}(\text{polly}), \text{penguin}(\text{polly}), \text{flies}(\text{tweety}), \text{flies}(\text{polly}), 
\text{not penguin(\text{tweety})}, \text{not ab(\text{tweety})}, \text{not ab(\text{polly})}, 
\text{not ~bird(\text{tweety})}, \text{not ~bird(\text{polly})}, \text{not ~penguin(\text{polly})}, 
\text{not ~penguin(\text{tweety})}, \text{not ~flies(\text{tweety})}, 
\text{not ~flies(\text{polly})}, \text{not ~ab(\text{tweety})}, \text{not ~ab(\text{polly})} \}.
\]
Picking the LP-literals from \( S^+ \) that are relevant to \( \text{flies}(\text{polly}) \) and are involved in \( P \cup L \), we get the following integrity constraint.
\[
\text{not ab}(\text{polly}) \rightarrow \text{bird}(\text{polly}), \text{penguin}(\text{polly}), \text{flies}(\text{polly}), \text{not ab}(\text{polly}).
\]
Moving \( \text{not ab}(\text{polly}) \) to the head, we get
\[
\text{ab}(\text{polly}) \leftarrow \text{bird}(\text{polly}), \text{penguin}(\text{polly}), \text{flies}(\text{polly}).
\]
Since \( \text{flies} \) strongly and negatively depends on \( ab \), dropping \( \text{flies}(\text{polly}) \), we get
\[
\text{ab}(\text{polly}) \leftarrow \text{bird}(\text{polly}), \text{penguin}(\text{polly}).
\]
Generalizing the rule, we get \( R = \)
\[
\text{ab}(x) \leftarrow \text{bird}(x), \text{penguin}(x).
\]
It is easy to check that \( P \cup R \) is consistent and \( P \cup R \not= \text{flies}(\text{polly}) \). \( R \) can be furthur simplified as the following, in view of (26):
\[
\text{ab}(x) \leftarrow \text{penguin}(x).
\]
As in the case of learning from positive examples, if \( P \) has multiple answer sets, a rule \( R \) is learnt with respect to each answer set \( S \) of \( P \). If we are given a set of negative examples \( E \) instead of just a literal \( L \), we apply the same learning approach for each literal \( L \in E \), considering the same background program \( P \) for all \( L \in E \).

We will see that some of the problems with respect to learning from positive examples are carried over to learning from negative examples.

### 4.1. Premature Generalization

**Example 8:** Consider the following background program \( P \) and the negative example \( L = \text{flies}(\text{polly}) \):

\[
\begin{align*}
\text{flies}(x) & \leftarrow \text{bird}(x), \text{not ab}(x). \\
\text{bird}(x) & \leftarrow \text{nightingale}(x). \\
\text{nightingale}(\text{polly}).
\end{align*}
\]

This program is obtained from the background program in Example 7 by just replacing \( \text{penguin} \) with \( \text{nightingale} \). The negative example \( \text{flies}(\text{polly}) \) might be a result of \( \text{polly} \) being crippled. However, given this negative example, the rule learnt using the approach defined earlier is:
\[
\text{ab}(x) \leftarrow \text{nightingale}(x).
\]

This rule when added to the program \( P \) will imply that every nightingale is abnormal and as a result cannot fly. However, this is not a serious problem as in order to avoid this, one can first train the learning program using positive examples - which is \( \text{crippled}(\text{polly}) \) in this case. However, this is not sufficient to handle certain drawbacks of this approach, as we will see in the following subsection.

### 4.2. Unintuitive dependencies in the learnt rule

**Example 9:** Consider Example 8 with the addition of the following rule to the background program \( P \):

\[
\text{crippled}(\text{polly}).
\]

The rule learnt in this case is:
\[
\text{ab}(x) \leftarrow \text{crippled}(x), \text{nightingale}(x). \tag{27}
\]

Though this rule is acceptable, the rule indicates that a bird being abnormal depends on the bird being a nightingale, which is unintuitive in the context of the program since nightingales generally fly. Ideally, we would want to learn the rule:
\[
\text{ab}(x) \leftarrow \text{crippled}(x).
\]

This problem is related to the problem resulting from not following a level based learning in case of learning from positive examples (section 3.2). However, the solution presented in section 3.2 is not suitable for learning from negative examples. This is because in case of learning from negative examples, we want to learn rules defining why an object is not related to a concept (which is usually done by finding conditions
under which the object is abnormal, i.e., belongs to predicate ab) and the reason for such non-existence of relationship is generally not dependent on the level of the concept. For example, consider the following rule learnt in Example 7:

\[ ab(x) \leftarrow \text{penguin}(x). \]

In the above rule, we can see that the property ab depends on penguin, which is intuitive as penguins cannot fly. However, in (27), the dependency of ab on nightingale is unintuitive. As a result, if polly is a penguin and molly is a nightingale, polly is abnormal and molly is not. On the other hand, one would naturally assume nightingale and penguin to be of the same level as both describe types of birds. So, assigning ab to a lower level than nightingale and penguin will work for Example 9 but will not work for Example 7.

5. Soundness and Completeness

5.1. Soundness

In this subsection, we present the soundness results from (Sakama, 2005). These results are still valid in view of the improvements presented in this paper.

Theorem 1: Let \( P \) be a background program and \( L \) be a ground literal representing a positive example. Let \( P \) have answer sets \( S_1, \ldots, S_n \) and let \( R_1, \ldots, R_n \) be the respective rules learnt. If each \( R_i \) is negative-cycle-free and \( \text{pred}(L) \) does not occur in \( P \), then \( P \cup \{ R_1, \ldots, R_n \} \models L \).

Consider the following background program \( P \) and the positive example \( L = r(a) \):

\[
\begin{align*}
p(x) & \leftarrow \text{not } q(x). \\
q(x) & \leftarrow \text{not } p(x).
\end{align*}
\]

Here, \( P \) has 2 answer sets \( S_1 = \{ p(a) \} \) and \( S_2 = \{ q(a) \} \). The rules learnt from \( P \) and \( L \) are

\[
R_1: r(x) \leftarrow p(x), \text{not } q(x).
\]

and

\[
R_2: r(x) \leftarrow q(x), \text{not } p(x).
\]

corresponding to \( S_1 \) and \( S_2 \) respectively. Note here that both the rules learnt are negative-cycle-free and \( r \) does not occur in \( P \). It is easy to check that \( P \cup \{ R_1, R_2 \} \models L \) holds.

Theorem 2: Let \( P \) be a background program and \( L \) be a ground literal representing a negative example. Let \( P \) have answer sets \( S_1, \ldots, S_n \) and let \( R_1, \ldots, R_n \) be the respective rules learnt where \( R_i \theta_i = K \leftarrow \Gamma''_i \) according to the last step in the learning algorithm. If \( P \cup \{ R_1 \theta_1, \ldots, R_n \theta_n \} \models R_i \) for all \( i \in \{ 1, \ldots, n \} \) and \( P \cup \{ R_1, \ldots, R_n \} \models \text{not } L \), then \( P \cup \{ R_1, \ldots, R_n \} \models L \).

Theorem 3: Let \( P \) be a background program that has exactly one answer set and \( \{ L_1, \ldots, L_n \} \) be a set of ground literals representing positive examples. Let \( R_i \) be the rule learnt from \( P \) and \( L_i \). If each \( R_i \) is negative-cycle-free and \( \text{pred}(L) \) does not occur in \( P \), then \( P \cup \{ R_1, \ldots, R_n \} \models L_i \) for all \( i \in \{ 1, \ldots, n \} \).

Consider the following background program \( P \) and the set of positive examples \( E = \{ \text{flies}(tweety), \text{not } \text{flies}(polly) \} \):

\[
\begin{align*}
\text{bird}(x) & \leftarrow \text{penguin}(x). \\
\text{bird}(tweety). \\
\text{penguin}(polly).
\end{align*}
\]

The rules learnt from \( P \) and \( E \) are

\[
R_1: \text{flies}(x) \leftarrow \text{bird}(x), \text{not } \text{penguin}(x).
\]

and

\[
R_2: \text{not } \text{flies}(x) \leftarrow \text{bird}(x), \text{penguin}(x).
\]

Theorem 4: Let \( P \) be a background program that has exactly one answer set and \( \{ L_1, \ldots, L_n \} \) be a set of ground literals representing negative examples. Let \( R_i \) be the rule learnt from \( P \) and \( L_i \) where \( R_i \theta_i = K \leftarrow \Gamma''_i \) according to the last step in the learning algorithm. If \( P \cup \{ R_1 \theta_1, \ldots, R_n \theta_n \} \models R_i \) and \( P \cup R_i \) is consistent for all \( i \in \{ 1, \ldots, n \} \) and if \( \text{pred}(K_j) \) does not depend on \( \text{pred}(K_j) \) in \( P \cup \{ R_1, \ldots, R_n \} \) for all \( i, j \in \{ 1, \ldots, n \} \), then \( P \cup \{ R_1, \ldots, R_n \} \models L_i \) for all \( i \in \{ 1, \ldots, n \} \) if \( P \cup \{ R_1, \ldots, R_n \} \) is consistent.

5.2. Completeness

We say that a learning algorithm is complete with respect to a positive example (resp. a negative example) \( L \) if it produces every rule \( R \) satisfying \( P \cup R \models L \) (resp. \( P \cup R \not\models L \)). The algorithms presented in (Sakama, 2005) are incomplete with respect to both positive and negative examples. In general, there can be many rules that justify a positive/negative example. However, it is important to find only those rules that are meaningful. For example, in Example 1, the following rule
also justifies $L$:

$$flies(tweety) ← bird(polly).$$

However, it is clear that this rule is not meaningful in the context of the example. The approach to learning reviewed and improved in this paper targets to find one meaningful rule that explains the example. This can be a disadvantage in some cases. For example, consider the following background program $P$ and the positive example $L = flies(tweety)$:

$$bird(x) ← nightingale(x).$$

$$bird(x) ← penguin(x).$$

$$bird(x) ← ostrich(x).$$

$$nightingale(tweety).$$

$$penguin(\text{polly}).$$

$$\text{ostrich}(\text{alex}).$$

The rule learnt from $P$ and $L$ is

$$flies(x) ← nightingale(x), \text{bird}(x), \text{not } penguin(x), \text{not } ostrich(x). \quad (28)$$

However, the following 2 rules can be considered more appropriate in the context of this example:

$$flies(x) ← nightingale(x).$$

$$flies(x) ← bird(x), \text{not } penguin(x), \text{not } ostrich(x).$$

The learning algorithms also try to learn rules with fewer unintuitive dependencies. In example 4, we noted that the learnt rule contains unintuitive dependencies: the rule learnt in example 4 ((16)) implies that a bird being able to walk depends on the bird being able to sing. In general it is very difficult, if not impossible, to avoid all such unintuitive dependencies. The notions of relevance and involvement defined in (Sakama, 2005) and reproduced in this paper specify conditions under which some of the unintuitive dependencies can be detected.

6. Mixed Learning

Mixed Learning deals with learning from both positive and negative examples. According to (Sakama, 2005), when sets of positive and negative examples are given, rules are learnt from the positive examples first and then from negative examples. In Example 8, we mentioned that the premature generalization can be avoided by first learning the rule from the positive example crippled(polly) and then from the negative example flies(polly). Below, we show how to accomplish this.

Example 10: Consider the following background program $P$:

$$flies(x) ← bird(x), \text{not } ab(x).$$

$$bird(x) ← nightingale(x).$$

$$\text{bird}(tweety),$$

$$\text{nightingale}(\text{polly}).$$

Consider first, the positive example crippled(polly) where crippled is at the same level as nightingale, and is at a lower level than bird and flies. Using the level based learning defined in section 3.2 and the approach to handle premature generalization, the rule learnt is:

$$\text{crippled}(\text{polly}). \quad (29)$$

Now, considering the background program $P \cup (29)$ and the negative example flies(polly), the rule learnt is (27).

As we saw in the above example, it is possible to learn successfully from both positive and negative examples and that this approach also avoids some of the problems with learning from negative examples only. However, learning from both positive and negative examples is not always successful. The following example is from (Sakama, 2005).

Consider the following background program $P$:

$$q(x) ← r(x).$$

$$q(a).$$

$$q(b).$$

$$r(c).$$

Consider the positive example $L = p(a)$. The rule learnt from the positive example is $R =$

$$p(x) ← q(x), \text{not } r(x).$$

Here the required condition $P \cup R \models L$ holds. Now, consider the negative example $L' = p(b)$. The rule learnt from the negative example using the background program $P \cup R$ is $R' =$

$$r(x) ← q(x).$$

Again, the required condition $P \cup R \cup R' \not\models L'$ holds. However, $P \cup R \cup R' \not\models L$ also holds which implies that the final program does not justify the positive example $L$. 
7. Nonmonotonicity

The result in the previous section that \( P \cup R \models L \) but \( P \cup R \cup R' \not\models L \) is due to the non-monotonicity of the answer set semantics. This case does not arise with learning rules under monotonic logics where, \( P \cup R \cup R' \models L \) if \( P \cup R \models L \) for all \( R' \). Though this monotonicity property is preferable during learning, as mentioned in the introduction, using monotonic semantics for reasoning is not suitable for representing and reasoning in the presence of incomplete information. A better approach would be to define learning algorithms based on non-monotonic semantics, but yet maintaining the monotonicity property as explained above. The following theorems provide conditions under which the learning algorithms in (Sakama, 2005) maintain the monotonicity property. These theorems still hold in view of the improvements presented in this paper.

Theorem 5: Let \( P \) be a program with exactly one answer set, and \( L_1 \) and \( L_2 \) be ground literals representing positive examples such that \( \text{pred}(L_1) \) and \( \text{pred}(L_2) \) do not occur in \( P \). If a negative-cycle-free rule \( R_1 \) is obtained from \( P \) and \( L_1 \) and a negative-cycle-free rule \( R_2 \) is obtained from \( P \cup R_1 \) and \( L_2 \), then \( P \cup R_1 \cup R_2 \models L_1 \land L_2 \).

Theorem 6: Let \( P \) be a program with exactly one answer set, and \( L_1 \) and \( L_2 \) be ground literals representing negative examples. If

- \( R_1 \) is the rule learnt from \( P \) and \( L_1 \), \( R_1 \theta_1 = K_1 \leftarrow \Gamma''_1 \) according to the last step in the learning algorithm, \( P \cup R_1 \theta_1 \models R_1 \), \( P \cup R_1 \) has exactly one answer set,

- \( R_2 \) is the rule learnt from \( P \cup R_1 \) and \( L_2 \), \( R_2 \theta_2 = K_2 \leftarrow \Gamma''_2 \) according to the last step in the learning algorithm, \( P \cup R_1 \cup R_2 \theta_2 \models R_2 \), \( P \cup R_1 \cup R_2 \) has exactly one answer set and

- \( \text{pred}(K_j) \) does not depend on \( \text{pred}(K_j) \) for \( j \in \{1, 2\} \) in \( P \cup R_1 \cup R_2 \),

then \( P \cup R_1 \cup R_2 \not\models L_1 \lor L_2 \).

8. Computational Complexity

Answer set solvers are tools that compute answer sets of extended logic programs considered in this paper. In recent times, there has been remarkable progress in computing answer sets efficiently. Some of the popular answer set solvers are: SModels\(^1\), DLV\(^2\), CModels\(^3\), Clasp and Clingo\(^4\). CModels computes the answer sets by formulating the problem of computing answer sets as a satisfiability problem (Lin & Zhao, 2002), and then calling SAT solvers to compute the models. The other solvers mentioned above compute answer sets directly. However, some of them, such as Clasp and Clingo, use techniques similar to the ones used in SAT solvers. The problem of computing answer sets of extended logic programs considered in this paper is NP-complete. However, in several cases, the answer sets are computed in polynomial time by the modern answer set solvers. Since the programs considered are function-free, the expansion set can also be computed in finite amount of time. The rest of the algorithm takes only polynomial time to compute the learnt rule.

9. Related Work

Apart from (Sakama, 2001) and (Sakama, 2005), there is some work done in the field of ILP that deals with learning nonmonotonic logic programs. (Bain & Muggleton, 1992) introduce an algorithm called closed world specialization. In that algorithm, monotonic rules satisfying positive examples are first constructed and they are subsequently specialized by incorporating negation-as-failure literals in the bodies. (Inoue & Kudoh, 1997) and (Lamma et al., 2000) introduce algorithms for learning extended logic programs. They also divide the process of learning nonmonotonic logic programs into two steps: producing monotonic rules by an ordinary induction algorithm for Horn ILP and then specializing them by introducing negation-as-failure literals in the bodies. However, these algorithms are based on Horn ILP and have problems such that once nonmonotonic rules are learnt and incorporated into the background knowledge, Horn induction algorithms cannot be applied anymore to the new (nonmonotonic) background knowledge. (Nicolas & Duval, 2001) present and algorithm for learning a default theory from a set of positive/negative examples. Since an extended logic program is viewed as a default theory, the algorithms presented in (Sakama, 2005) and improved in this paper can also be considered as methods for learning default theories.

An important difference between the approaches in [(Inoue & Kudoh, 1997), (Lamma et al., 2000), (Nicolas & Duval, 2001)] and (Sakama, 2005) is with respect to the treatment of negative examples. (Inoue

\(^1\)http://www.tcs.hut.fi/Software/smodels/
\(^2\)http://www.dbai.tuwien.ac.at/proj/dlv/
\(^3\)http://www.cs.utexas.edu/~tag/cmodels/
\(^4\)http://potassco.sourceforge.net/
Learning definitions in Answer Set Programming

and Kudoh, 1997), (Lamma et al., 2000) and (Nicolas & Duval, 2001) represent negative examples by negative literals and define the problem of learning from negative example \( L \) as finding a set of default rules \( H \) satisfying \( P \cup H \models L \). In contrast, (Sakama, 2005) allows both positive and negative literals to be given as negative examples and defines the problem of learning from a negative example \( L \) as computing a rule \( R \) such that \( P \cup R \not\models L \).

(Seitzer, 1997) proposes a system called \textit{INDED} that consists of a deductive engine which computes the stable models or the well-founded model of a background normal logic program, and an inductive engine which empirically constructs hypotheses using a generic top-down approach. The top-down algorithm is an ordinary learning algorithm used in Horn ILP.

(Otero, 2001) provides an approach for learning in \textit{normal logic programs} (extended logic programs without classical negation \( \neg \)) under the stable model semantics. In contrast to the approach in (Sakama, 2005), he considers minimal and most specific solutions by introducing facts to a program. For example, in example 1, \( R = \text{flies(tweety)}. \)

is the most specific solution that satisfies \( P \cup R \models L \). His learning approach however considers learning from sets containing both positive and negative examples, where as the approach in (Sakama, 2005) (and this paper) only considers learning from sets of positive and negative examples (i.e., a set cannot contain both positive and negative examples).

10. Conclusion

Logic Programming provides powerful frameworks for non-monotonic reasoning. It is however, very difficult, if not impossible, for humans to encode all knowledge required to perform various kinds of reasoning. Thus arises the need for learning in the framework of logic programming. There has been considerable progress made in the area of Inductive Logic Programming. However, as mentioned in the introduction, most of the approaches in ILP focus on learning classical clausal theories which are not expressive enough to represent and reason about domains in the presence of incomplete knowledge. Answer Set Programming is more expressive and is one of the most popular forms of logic programming. However, there has not been much work done with respect to learning in the framework of ASP. In this paper, we considered some problems with the approach presented in (Sakama, 2005) and also discussed ways to avoid some of them. We also showed that the theoretical results presented in (Sakama, 2005) still hold in view of the improvements presented in this paper. As part of future work, we plan to provide a comprehensive framework based on the approach in (Sakama, 2005) for learning concepts/definitions in ASP. We plan to use the results presented in this paper for that purpose.

References


Lin, F., & Zhao, Y. (2002). Assat: Computing answer sets of a logic program by sat solvers. \textit{AAAI/IAAI} (pp. 112–).


