Learning Extended Logic Programs

Katsumi Inoue
Department of Electrical
and Electronics Engineering
Kobe University
Rokkodai, Nada-ku, Kobe 657, Japan
inouefieedept.kobe-u.ac.jp

Yoshimitsu Kudoh
Division of Electronics
and Information Engineering
Hokkaido University
N-13 W-8, Sapporo 060, Japan
kudoOdb.huee.hokudai.ac.jp

Abstract

This paper presents a method to generate nonmonotonic rules with exceptions from positive/negative examples and background knowledge in Inductive Logic Programming. We adopt extended logic programs as the form of programs to be learned, where two kinds of negation—negation as failure and classical negation—are effectively used in the presence of incomplete information. While default rules axe generated as specialization of general rules that cover positive examples, exceptions to general rules are identified from negative examples and are then generalized to rules for cancellation of defaults. We implemented the learning system LELP based on the proposed method. In LELP, when the numbers of positive and negative examples are very close, either parallel default rules with positive and negative consequents or nondeterministic rules are learned. Moreover, hierarchical defaults can also be learned by recursively calling the exception identification algorithm.

1 Introduction

Inductive logic programming (ILP) is a research area which provides theoretical frameworks and practical algorithms for inductive learning of relational descriptions in the form of logic programs [12, 10, 4]. Most previous work on ILP consider definite Horn programs or classical clausal programs in the form of logic programs to be learned. However, research work on knowledge representation in AI has shown that such monotonic programs are not adequate to represent our commonsense knowledge including notions of concepts and taxonomies. In this respect, there have been much work on *nonmonotonic reasoning* in AI. To learn default rules or concepts in taxonomic hierarchy, we thus need a learning mechanism that can deal with nonmonotonic reasoning.

On the other hand, recent advances on theories of logic programming and nonmonotonic reasoning have revealed that logic programs with *negation as failure* (NAF) is an appropriate tool for knowledge representation [3]. *Normal logic programs* (NLPs) are the class of programs in which NAF is allowed to appear freely in bodies of rules. NLPs are useful not only to represent default rules or rules with exceptions but also to write shorter and clearer programs than definite programs in many cases [5]. Learning NLPs has recently been considered in such as [2, 15, 5, 11].

While learning NLPs is an important step towards a better learning tool, there is still a limitation as a knowledge representation tool: NLPs do not allow us to deal directly with incomplete information [8]. NLPs automatically applies the *closed world assumption* (CWA) to all predicates, and any query is answered either *yes* or *no,* in which the latter negative answer is the result of CWA. In the context of inductive concept learning, the automatic application of CWA is not appropriate in the presence of both positive and negative examples. Positive examples represent instances of the target concept, while negative examples are non-instances. By CWA other objects are assumed non-instances, but then the role of negar tive examples is not clear because it is as if we supply a complete classification of all objects. This causes the paradox pointed out by De Raedt and Bruynooghe [6]: if everything is known, why should we still learn something? In the real world, we may not know whether some objects are positive or negative. But such incomplete information cannot be represented by NLPs.

To overcome the above problem of NLPs, we propose in this paper a new learning method which can deal with incomplete information in the form of *extended logic programs* (ELPs). ELPs are introduced by Gelfond and Lifschitz [8] to extend the class of NLPs by including *classical negation* (or explicit negation). The semantics of ELPs is given by the notion of *answer sets,* and is an extension of the *stable model semantics.* The answer to a ground query *A* is either *yes, no,* or *unknown,* depending on whether the answer set contains *A,* ¬, or neither. Using ELPs, the role of negative examples becomes clear, and any object not contained in either positive or negative examples is considered *unknown* unless the learned theory says that it must or must not be in that concept.

In this paper, we present a system, called **LELP** (Learning ELPs), to learn default rules with exceptions in the form of extended logic programs given incomplete positive and negative examples and background knowledge. LELP first generates candidate rules from positive examples (or negative examples if non-instances are much more than instances) and background knowledge in an ordinary ILP framework. Exceptions can be identified as negative examples (or positive examples if candidate rules have negative consequents) that are derived from the generated monotonic rules and background knowledge. Default rules with NAF are then computed by specializing candidate rules using the *open world specialization* (OWS) algorithm. This OWS algorithm is closely related to Bain and Muggleton's CWS algorithm [2], but works better in the three-valued semantics. Then, default cancellation rules are generated to cover exceptions using an ordinary ILP framework.

In the real world, it is not easy to know that a general default rule should have the positive or negative consequent. In LELP, it is determined according to the ratio of positive examples. Nevertheless, if it is still hard to know which is more general, LELP can generate nondeterministic rules in the context of the answer set semantics. Furthermore, by calling the OWS algorithm recursively, LELP can generate hierarchical default rules.

The rest of this paper is organized as follows. Section 2 outlines how our system LELP produces ELPs to learn simple default rules. Section 3 extends LELP to deal with complex concept structures with hierarchical exceptions. Section 4 presents related work, and Section 5 concludes the paper.

2 Learning Default Rules

This section shows how LELP learns default rules with exceptions. To clarify the underlying idea, we here consider a simple model that there are general rules and exceptions to them but there are no exceptions to exceptions. This model will be extended in Section 3.

The algorithm of LELP is summerized as follows.

Algorithm 2.1 $LELP1(E^+, E^-, BG, T1, T2, T3)$

```
Input: positive examples $E^+$, negative examples $E^-$,
        background knowledge $BG$
Output: default rules $T1$, exception rules $T2 \cup T3$
  1. According to the ratio of $E^+$ or $E^-$ to all objects,
     determine whether the learned general rule is positive or negative;
  2. Given $E^+$ (or $E^-$ if a negative rule is to be learned)
     and $BG$, generate general rules $T$ using an ordinary ILP technique;
  3. Given $T$, $E^+$, $E^-$ and $BG$, compute default rules
     $T1$ and exceptions $AB$ using the OWS algorithm;
     Generate rules $T2$ deriving $E^-$ (or $E^+$) from $AB$ at
     the same time using an ordinary ILP technique;
  4. Given $AB$ and $BG$, generate default cancellation
     rules $T3$ using an ordinary ILP techniques.
```

In LELP, the input positive examples are represented as *positive literals*, and negative examples are denoted as *negative literals*. We allow *rules* in the form of ELPs in background knowledge.

2.1 Extended Logic Programs

Extended logic programs (ELPs) were introduced in [8] as a tool for reasoning in the presence of incomplete information. They are defined as sets of rules of the form

$$L_0 \leftarrow L_1, \ldots, L_m, \mathit{not}\, L_{m+1}, \ldots, \mathit{not}\, L_n$$

where L_i's ($0 \leq i \leq n$; $n \geq m$) are literals. Two kinds of negation appear in a program: *not* is the *negation as failure* operator, and $\neg$ is *classical negation*. Intuitively, the above rule can be read as: if $L_1, \ldots, L_m$ are believed and $L_{m+1}, \ldots, L_n$ are not believed then L_0 is believed. The semantics of ELPs are defined by the notion of *answer sets* [8], which are sets of ground literals representing possible beliefs. The class of ELPs are considered as a subset of *default logic* [14]: each rule of the above form in an ELP can be identified with the *default* of the form

$$\frac{L_1 \wedge \ldots \wedge L_m \;:\; \overline{L_{m+1}}, \ldots, \overline{L_n}}{L_0}$$

where $\overline{L}$ stands for the literal complementary to L. Then, each *answer set* is the set of atoms in an extension of the default theory. We say that a literal L is *entailed* by an ELP P if L is contained in every answer set of P. While we adopt the answer set semantics in this par per, other semantics for ELPs may be applicable to our learning framework with minor modification.

We call a rule having a positive literal in its head *positive rule,* and a rule having a negative literal in its head *negative rule.* In the following, we denote classical negation $\neg$as - and NAF *not* as \+ in programs.

The completeness and consistency of concept learning (see [10, 4] for instance) can be reformulated in the three-valued setting as follows. Let *BG* be an ELP as background knowledge, *E* a set of positive/negative literals as positive/negative examples, and Ra set of rules as hypotheses. *R* is *complete* with respect to *BG* and *E* if for every e € 25, e is entailed by *BG* U *R (R covers* e). *R* is *consistent* with *BG* and *E* if for any e € *E,* eis not entailed by *BGuR (R does not cover* e). Note here that positive examples are not given any higher priority than negative ones. Namely, *both positive and negative examples are to be covered by the learned rules that are consistent with background knowledge and examples.* Thus, we will learn both positive and negative rules: no CWA is assumed to derive non-instances (see also [6]).

Although both positive and negative rules are generated by LELP, each default rule for the target concept should be either positive or negative. In LELP, it is determined according to the ratio of positive examples to all objects. In the following, we assume that positive rule is learned as a general rule unless otherwise specified.

2.2 Generating General Rules

In Algorithm 2.1, given positive (resp. negative) examples *E* and background knowledge *BG,*LELP generates general rules *T* to cover every example in *E* using an ordinary ILP technique. We denote this part of algorithm as *GenRules(E,BG,T).* In generating positive (resp.

negative) rules, no negative (resp. positive) example is used to specialize rules. The specialization of general rules is performed in the OWS algorithm (Section 2.3).

We do not assume any particular learning algorithm for the implementation of an "ordinary" ILP technique in *GenRules*(E, BG, T). This part can be considered as a black box, and this paper is not concerned with the detail. In our real implementation, we used the notion of Plotkin's RLGG (relative least general generalization) with the bottom-up technique used in GOLEM [13]. When background knowledge contains rules with bodies, the *unfold* transformation in logic programming is also used to truncate literals in bodies of learned rules.

Example 2.1 Suppose that positive examples E^+ are:
{flies(1),flies(2),flies(3),flies(4)},
and that background knowledge BG is:
{bird(1),bird(2),bird(3),bird(4),bird(c),
(bird(X) :- pen(X)),pen(a),pen(b)}.
Then, *GenRules*(E^+, BG, T) generates:

```
flies(X) :- bird(X).
```

2.3 Specializing Rules using NAF

The general rules computed to cover the positive (resp. negative) examples by *GenRules(E,BG,T)* may also cover the complements of some of negative (resp. positive) examples. To specialize general rules, we propose the algorithm of *open world specialization* (OWS). The OWS algorithm is closely related to Bain and Muggleton's closed world specialization (CWS) [2]. Like CWS, OWS produce rules with NAF as default rules. Unlike CWS, however, OWS does not apply the closed world assumption (CWA) to identify non-instances of the target concept. In OWS, exceptions are identified as objects contained in negative examples (or positive examples if the general rule is negative) such that they are proved from the general rule with background knowledge and positive (or negative) examples.

In the following OWS algorithm, we assume here that each general rule in T is positive.

Algorithm 2.2 $OWS(T, E^+, E^-, BG, AB, T')$

```
Input: rules $T$, positive examples $E^+$, negative examples
        $E^-$, background knowledge $BG$
Output: default rules $T'$, set of exceptions $AB$
Let $T' := T$; $AB := \emptyset$;
for each $C_i = (H$ :- $B)$ in $T$ do:
    Find a literal $L$ such that $\overline{L} \in E^-$ and $L$ is entailed
    by $BG \cup \{C_i\} \cup E^+$ (we call $L$ an exception to $C_i$
    and $C_i$ a rule with exceptions);
    $\theta :=$ the answer substitution for $C_i$ in proving $L$;
    if $B$ contains the literal \+ $N$ then
        $AB := AB \cup \{N\theta\}$
    else $N := ab_i(V_1, \ldots, V_n)$,
        where $\{V_1, \ldots, V_n\}$ is the domain of $\theta$;
        $T' := (T' \setminus \{C_i\}) \cup \{(H$ :- $B$, \+$N)\}$;
        $AB := AB \cup \{N\theta\}$.
```

In the real implementation of LELP, we used Prolog's top-down proof is used to get answer substitutions.

Example 2.2 (cont. from Example 2.1)
BG, E^+: the same as Example 2.1,
T = {(flies(X) :- bird(X))},
E^- = {-flies(a),-flies(b),-flies(c)}.
Then, the exceptions to $C1$ = (flies(X) :- bird(X)) are computed as {flies(a),flies(b),flies(c)}. In this case, substitutions θ's are X/a, X/b, X/c, and N is ab1(X). Hence,
T' = {(flies(X) :- bird(X), \+ab1(X))},
AB = {ab1(a),ab1(b),ab1(c)}.

2.4 Negative Rules for Exceptions

Since we use OWS, we need rules to derive negative examples (or positive examples if the default rule is negative). Given negative examples E^- (resp. positive examples E^+) and the set AB of exceptions, LELP generates negative (resp. positive) rules R to derive exceptions as *GenRules*(E^-, AB, R) (resp. *GenRules*(E^+, AB, R)).

In the bird example, such a rule is generated as:

```
-flies(Y) :- ab1(Y).
```

2.5 Cancellation Rules

In the OWS algorithm, the set *AB* of exceptions is output as a set of ground atoms. However, if exceptions have some common properties, this expression is not informative and *rules about exceptions* are useful. These rules work as *default cancellation rules.*

After applying OWS, each exception is in the form of ground atom whose predicate is ab_i Rules about exceptions have such abnormal predicates in their heads and are results of generalizations of some abnormal atoms. When such a common rule cannot be generated or there are some exceptions that cannot be covered by such a rule, those exceptions are left as they are.

Since exceptions are not anticipated in general, rules about exceptions should be used to derive only exceptions. In fact, exceptions are usually minimized in nonmonotonic reasoning. To this end, we apply a limited form of CWA here. If a rule about exceptions is too general, that is, it derives negative facts more than expected, it should be rejected. This test can be done easily using a bottom-up model generation procedure. The algorithm to generate rules about exceptions is as follows.

Algorithm 2.3 *Cancel*(AB, BG, R)

```
Input: set of exceptions $AB$, background knowledge $BG$
Output: default cancellation rules $R$
  1. GenRules$(AB, BG, T)$;
  2. For each $C \in T$, compute the set $L$ of $ab_i$ literals
     that are entailed by $BG \cup \{C\} \cup AB$;
     if $L \supset AB$ then $R := T \setminus \{C\}$
                  else $R := T$.
```

Example 2.3 (cont. from Example 2.2)
BG: the same as Example 2.1,
AB = {ab1(a),ab1(b),ab1(c)},
E = {flies(1),flies(2),flies(3),flies(4)}.
Suppose that *GenRules* outputs T that contains rules (ab1(X) :- pen(X)) and (ab1(Y) :- bird(Y)).

Here, the rule (ab1(Y) :- bird(Y)) enables us to derive ab1(1), ab1(2), ab1(3), ab1(4) besides the set AB, so it is removed by Algorithm 2.3. The atom ab1(c) represents an exception that is not a penguin, and hence cannot be generalized. The final rules are:

$R = \{(\texttt{ab1(Z) :- pen(Z)}), \texttt{ab1(c)}\}$

While in this section we apply CWA to generate rules about exceptions in LELP, this assumption is not appropriate for exceptions to exceptions. We will extend the framework in this respect in Section 3.2.

2.6 Properties

Given consistent background knowledge BG and positive and negative examples $E = E^+ \cup E^-$, let R be the output hypotheses learned by LELP. Then, the next theorem is proved in [9].

Theorem 2.1 *R is complete with respect to BG and E, and is consistent with BG and E.*

2.7 Example

The LELP program is implemented in Prolog and is called by

lelp(Examples,Background_Knowledge,Result).

In Examples, atoms preceded by + represents positive examples, and those with - are negative examples. In the following example, if the ratio of positive (resp. negative) examples exceeds 50%, positive (resp. negative) rules are generated.

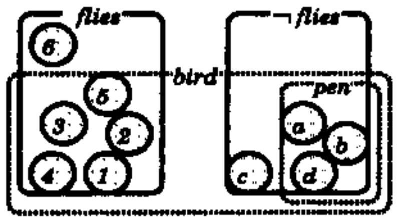

Figure 1: Bird Example

Example 2.4 (Bird Example Summary: Figure 1)

```
| ?- lelp([+flies(1),+flies(2),+flies(3),
        +flies(4),+flies(5),+flies(6),
        -flies(a),-flies(b),-flies(c),-flies(d)],
       [bird(1),bird(2),bird(3),bird(4),bird(5),
        bird(c),(bird(X) :- pen(X)),
        pen(a),pen(b),pen(d)],Rules).

RLGG/LL of flies(1) and flies(2) is
     flies(_4594):-bird(_4594)
Covered examples: flies(1), flies(2), flies(3),
     flies(4), flies(5)
RLGG/LL of -flies(a) and -flies(b) is
     -flies(_42885):-pen(_42885),bird(_42885)
Covered examples: -flies(a), -flies(b), -flies(d)
RLGG/LL of ab1(a) and ab1(b) is
     ab1(_85217):-pen(_85217),bird(_85217)
Covered examples: ab1(a), ab1(b), ab1(d)
RLGG/LL of -flies(a) and -flies(b) is
     -flies(_127580):-ab1(_127580)
Covered examples: -flies(a), -flies(b), -flies(c),
     -flies(d)
Unfolding of ab1(_410):-pen(_410),bird(_410)
with bird(_410):-pen(_410) is
     ab1(_410):-pen(_410)

Running Time: 280 (msec)

Rules =
[(flies(A):-bird(A),\+ab1(A)),flies(6),
 (-flies(B):-ab1(B)),
 (ab1(C):-pen(C)),ab1(c)] ?
```

3 Extension

In this section, we extend LELP to learn more complex concept structures.

3.1 Nondeterministic Rules

When the number of positive examples is close to that of negative examples, it is difficult to judge whether the general rule should be positive or negative. Two solutions can be considered to this problem: (1) *parallel default rules,* and (2) *nondeterministic rules.* Parallel default rules are generated when exceptions exist for both positive and negative rules in parallel (e.g., mammals normally do not fly except bats, and birds normally fly except penguins). Nondeterministic rules are generated when some object is proved to be positive and negative by a program such that a contradiction occurs. An extension of Algorithm 2.1 is shown in Section 3.2, where hierarchical defaults can also be learned.

In the following example, if the ratio of positive examples is between 40% and 60%, parallel default rules or nondeterministic rules are generated.

Example 3.1 (Learning Nondeterministic Rules)

```
| ?- lelp([+flies(1),+flies(2),+flies(3),+flies(4),
       -flies(5),-flies(6),-flies(7),-flies(8)],
      [bird(1),bird(2),bird(3),bird(4),
       bird(5),bird(6),bird(7),bird(8)],Rules).
```

Here, the ratio of positive examples is 50%, and hence both positive and negative rules are generated. If we use the previous method, the following rules are computed:

```
(flies(A):-bird(A),\+ab1(A)),
(-flies(B):-ab1(B)),
ab1(5),ab1(6),ab1(7),ab1(8),
(-flies(C):-bird(C),\+ab2(C)),
(flies(D):-ab2(D)),
ab2(1),ab2(2),ab2(3),ab2(4)
```

The above rules look like correct, but the bodies of both rules are the same as bird(X) except the ab$_i$ literals. Then, if bird(9) is added to background knowledge, neither ab1 nor ab2 is proved so that a contradiction occurs. In this case, the nondeterministic rules are generated:

```
(flies(A):-bird(A),\+ab1(A),\+ -flies(A)),
(-flies(B):-ab1(B)),ab1(5),ab1(6),ab1(7),ab1(8),
(-flies(C):-bird(C),\+ab2(C),\+flies(C)),
(flies(D):-ab2(D)),ab2(1),ab2(2),ab2(3),ab2(4)
```

This form of nondeterministic rules are shown in [3] to represent default rules in ELPs. For the added ground term 9, the ground instances of general rules are:

```
flies(9) :- bird(9), \+ab1(9), \+ -flies(9).
-flies(9) :- bird(9), \+ab2(9), \+flies(9).
```

Hence, for the bird 9, two answer sets exist, one concluding `flies(9)` and the other `-flies(9)`.

Here are two examples of Nixon Diamond.

Example 3.2 (Pacifist Nixon)

```
| ?- lelp([+pacifist(c),+pacifist(d),
         -pacifist(a),-pacifist(b),+pacifist(nixon)],
        [republican(a),republican(b),
         republican(nixon),quaker(nixon),
         quaker(c),quaker(d)],Rules).
```

The result shows that parallel rules are generated in which only Nixon is an exception of republicans:

```
(pacifist(A):-quaker(A)),
(-pacifist(B):-republican(B),\+ab1(B)),
ab1(nixon)
```

Example 3.3 (Nixon's Trouble)

```
| ?- lelp([+pacifist(c),+pacifist(d),
         -pacifist(a),-pacifist(b)],
        [republican(a),republican(b),
         republican(nixon),quaker(nixon),
         quaker(c),quaker(d)],Rules).
```

The result implies two answer sets for Nixon, and neither `pacifist(nixon)` nor `-pacifist(nixon)` is entailed:

```
(pacifist(A):-quaker(A),\+ -pacifist(A)),
(-pacifist(B):-republican(B),\+pacifist(B))
```

3.2 Hierarchical Defaults

Here, we further extend LELP to deal with hierarchical structures of concepts and exceptions. First, we modify Algorithm 2.3 to generate default cancellation rules.

Algorithm 3.1 *Cancel2*(AB, BG, R)

```
Input: set of exceptions $AB$, background knowledge $BG$
Output: default cancellation rules $R$
  1. $GenRules(AB, BG, T)$;
  2. For each $C \in T$, compute the set $L$ of ab_i literals
     that are entailed by $BG \cup \{C\} \cup AB$;
     if $|L - AB| \geq |AB|$ then $R := T \setminus \{C\}$
                         else $R := T$.
```

The new condition $|L - AB| \geq |AB|$ in Step 3 above replaces the CWA condition $L \supset AB$ in Algorithm 2.3, and represents the *monotone assumption* that "in every level of the hierarchy, the number of exceptions is less than that of instances with default properties".

To learn hierarchical default rules, after removing over-general rules in Algorithm 3.1, we need to call algorithms of *OWS* and *Cancel* recursively. The procedure stops when there are no more exceptions or no more objects to be generalized. The extended algorithm LELP2 is as follows. In the following, we denote sets of opposite examples as E_1, E_2 instead of positive examples E^+ and negative examples E^-.

Algorithm 3.2 *LELP2*(E_1, E_2, BG, R)

```
Input: positive examples $E_1$, negative examples $E_2$,
       background knowledge $BG$
Output: $R := R_3 \cup R_4 \cup R_5 \cup R_6$
  1. According to the ratio of $E_1$ (or $E_2$) to all objects,
     determine to learn either (1) default rules for $E_1$,
     (2) default rules for $E_2$, (3) parallel default rules, or
     (4) nondeterministic rules;
     In case of (1), perform Steps 2, 4, 7,
     put $R_4 = R_6 = \emptyset$, and return;
     In case of (2), perform Steps 3, 5, 8,
     put $R_3 = R_5 = \emptyset$, and return;
     Otherwise, execute every step after 2 in order;
  2. $GenRules(E_1, BG, R_1)$;
  3. $GenRules(E_2, BG, R_2)$;
  4. $OWS(R_1, E_1, E_2, BG, AB_1, R_3)$;
  5. $OWS(R_2, E_2, E_1, BG, AB_2, R_4)$;
  6. If there are contradictory rules in $R_1$ and $R_2$,
     transform $R_3$ and $R_4$ into nondeterministic rules,
     put $R_5 = R_6 = \emptyset$, and return;
  7. if $AB_1 \neq \emptyset$ then $ABs(BG, AB_1, E_1, E_2, R_5)$;
                    else $R_5 := \emptyset$;
  8. if $AB_2 \neq \emptyset$ then $ABs(BG, AB_2, E_2, E_1, R_6)$;
                    else $R_6 := \emptyset$.
```

Algorithm 3.3 *ABs*(BG, AB_1, E_1, E_2, R)

```
  1. $Cancel2(AB_1, BG, R_1)$;
  2. $OWS(R_1, E_2, E_1, BG, AB_2, R_2)$;
  3. if $AB_2 \neq \emptyset$ then $ABs(BG, AB_2, E_2, E_1, R_3)$;
                    else put $R := R_1$, and return;
  4. $GenRules(AB_2, BG, R_4)$;
     put $R := R_2 \cup R_3 \cup R_4$, and return.
```

Example 3.4 (Learning Hierarchical Defaults)

```
| ?- lelp([-flies(1),-flies(2),+flies(3),+flies(4),
         +flies(5),-flies(6),-flies(7),-flies(8),
         -flies(9),-flies(10),-flies(11),-flies(12)],
        [pen(1),pen(2),bird(3),bird(4),bird(5),
         (bird(X) :- pen(X)),animal(6),animal(7),
         animal(8),animal(9),animal(10),animal(11),
         animal(12),(animal(X) :- bird(X))],Rules).
```

Here, the ratio of positive examples is 25%, and hence the negative rule `(-flies(A):-animal(A))` is firstly generated. Using OWS, the set of exceptions in the first level {`ab1(3)`, `ab1(4)`, `ab1(5)`} and the default rule `(-flies(A):-animal(A),\+ab1(A))` are then computed. This set of exceptions is generalized to the cancellation rule `(ab1(C):-bird(C))` and the positive rule for exceptions `(flies(B):-ab1(B))` are also generated. Nextly, after applying OWS to the cancellation rule, the exceptions in the second level {`ab2(1)`, `ab2(2)`} and the default rule are computed, and then its cancellation rule and the negative rule for exceptions are generated. At this point, no more exception is left and the algorithm stops. The final `Rules` are as follows.

```
(-flies(A):-animal(A),\+ab1(A)),
(flies(B):-ab1(B)),
(ab1(C):-bird(C),\+ab2(C)),
(-flies(D):-ab2(D)),
(ab2(E):-pen(E))
```

4 Related Work

Bain and Muggleton's CWS algorithm [2] has been applied to non-monotonic versions of CIGOL and GOLEM in [l] and a learning algorithm that can acquire hierarchical programs in [15] CWS produces default rules with NAF in stratified NLPs. Since CWS is based on CWA in the two-valued setting, it regards every ground atom that is not contained in an intended model as an exception. In LELP, on the other hand, OWS is employed instead of CWS, and incomplete information can be represented in ELPs with the three-valued semantics.

TRACY^{not} by Bergadano *et al.* [5] learns stratified NLPs using trace information of SLDNF derivations. Since this system needs the hypothesis space in advance, it does not invent a new predicate like ab_iexpressing exceptions, and hence seems more suitable for learning rules with negative knowledge and CWA rather than learning defaults. Martin and Vrain [ll] use the three-valued semantics for NLPs in their inductive framework. Since they do not adopt ELPs, CWA is still employed and two kinds of negation are not distinguished.

While no previous work adopts full ELPs in the form of learned programs, a limited form of classical negation has been used in [6, 7]. De Raedt and Bruynooghe [6] firstly discussed the importance of the three-valued semantics in ILP. However, since they did not allow NAF, an explicit list of exceptions is necessary for each rule, which causes the *qualification problem* in AL Wrobel [16] also used exception lists to specialize over-general rules, but their underlying language is monotonic first-order. Dimopoulos and Kakas [7] propose a learning method that can acquire rules with hierarchical exceptions. They also do not use NAF to represent defaults, but adopt their own nonmonotonic logic. Moreover, using the approach of [7], one has to determine whether each negative information should be used in the usual specialization process or in the exception identification process. In our approach, such distinction can be clearly done by an appropriate usage of NAF and classical negation.

Finally, in any previous work, nondeterministic rules cannot be generated, and hence commonsense knowledge with multiple extensions cannot be learned.

5 Conclusion

This paper proposed new techniques to learn nonmonotonic rules with exceptions, and introduced the learning system LELP. Extended logic programs are adopted as program forms, in which two kinds of negation are effectively used in the presence of incomplete information. Default rules are generated using OWS, and their exceptions are then generalized to cancellation rules. LELP can also learn parallel/nondeterministic rules and hierarchical defaults within the three-valued semantics.

In this paper, we treated every explicit negative information as an exception to a positive hypothesis. In the real world, however, negative knowledge may often be irrelevant to the concepts to be learned. In this respect, a method of separation of noise from exceptions has been proposed in [15], Another approach is that we - may add information that each concept can have exceptions or not or that CWA can be applied or not. These extensions can easily be accommodated within LELP.

References

[1] Michael Bain. Experiments in non-monotonic first-order induction. In: [12], pages 423-436.

[2] Michael Bain and Stephen Muggleton. Non-monotonic learning. In: [12], pages 145-161.

[3] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming,* 19/20:73-148, 1994.

[4] Francesco Bergadano and Daniele Gunetti. *Inductive Logic Programming: Prom Machine Learning to Software Engineering.* MIT Press, 1996.

[5] F. Bergadano, D. Gunetti, M. Nicosia and G. Ruffo. Learning logic programs with negation as failure. In: Luc De Raedt, editor, *Proceedings of ILP-95,* pages 33-51, K.U. Leuven, 1995.

[6] Luc De Raedt and Maurice Bruynooghe. On negation and three-valued logic in interactive concept-learning. In: *Proceedings of ECAI '90,* pages 207-212, Pitman, 1990.

[7] Yannis Dimopoulos and Antonis Kakas. Learning non-monotonic logic programs: learning exceptions. In: Nada Lavrac and Stefan Wrobel, editors, *Proceedings of ECML-95,* pages 122-137, LNAI 912, Springer, 1995.

[8] Michael Gelfond and Vladimir Lifsitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing,* 9(3,4):365-385, 1991.

[9] Katsumi Inoue and Yoshimitsu Kudoh. Learning default rules in extended logic programs. Submitted for publication, 1997 (in Japanese).

[10] Nada Lavrac and Saso Dzeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1994.

[11] Lionel Martin and Christel Vrain. A three-valued framework for the induction of general logic programs. In: Luc De Raedt, editor, *Advances in Inductive Logic Programming,* pages 219-235, IOS Press, 1996.

[12] Stephen Muggleton, editor. *Inductive Logic Programming.* Academic Press, London, 1992.

[13] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In: [12], pages 281-298.

[14] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence,* 13:81-132, 1980.

[15] Ashwin Srinivasan, Stephen Muggleton and Michael Bain. Distinguishing exceptions from noise in non-monotonic learning. In: *Proceedings of ILP-92* ICOT, 1992.

[16] Stefan WrobeL On the proper definition of minimality in specialization and theory revision. In: Pavel B. Brazdil, editor, *Proceedings of ECML-93,*ages 65-82, LNAI 667, Springer, 1993.