Learning with Extended Logic Programs

Evelina Lamma and **Fabrizio Riguzzi**
DEIS, Università di Bologna,
Viale Risorgimento 2
40136 Bologna, Italy,
{elamma,friguzzi}@deis.unibo.it

Luís Moniz Pereira
Centro de Inteligência Artificial (CENTRIA),
Departamento de Informática,
Universidade Nova de Lisboa,
2825 Monte da Caparica, Portugal
lmp@di.fct.unl.pt

Abstract

We discuss the adoption of a three-valued setting for inductive concept learning. Distinguishing between what is true, what is false and what is unknown can be useful in situations where decisions have to be taken on the basis of scarce information. In a three-valued setting, we want to learn a definition for both the target concept and its opposite, considering positive and negative examples as instances of two disjoint classes. To this purpose, we adopt extended logic programs under a well-founded semantics as the representation formalism for learning. In this way, we are able to represent both the concept and its opposite and deal with incomplete or unknown information.

We discuss various approaches to be adopted in order to handle possible inconsistencies. Default negation is used to ensure consistency and to handle exceptions to general rules. Exceptions to a positive concept are identified from negative examples, whereas exceptions to a negative concept are identified from positive examples. Exceptions can be generalized, in their turn, by learning within a hierarchy of defaults.

Introduction

Most work on inductive concept learning considers a two-valued setting. In such a setting, what is not entailed by the learned theory is considered as false, by using the Closed World Assumption (CWA) (Reiter 1978). However, in practice, it is more often the case that we know with certainty the truth or falsity of a limited number of facts and we are not able to draw any conclusion on the remaining facts, because the information available is too scarce. As it has been pointed out in (De Raedt & Bruynooghe 1990), this is typically the case of an autonomous agent that incrementally gathers information from its surrounding world. The agent has to choose its actions on the basis of the knowledge it possesses and knowing that an action certainly leads to a failure is different from not knowing anything about its outcome. It will never try an action when it is sure of its negative effect, but it may try an action with an unknown outcome in situations where no other action can be taken or in order to expand its knowledge.

Therefore, for such an agent, it would be much better to be able to distinguish between what is certainly true, what is certainly false and what is unknown. To this purpose, the agent should adopt a three-valued setting and learn a definition for both the concept and its complement, using positive examples for the concept as negative examples for its complement and viceversa. The learned theory will then classify instances in three ways: instances covered by the positive definition are positive, instances covered by the negative definition are negative and instances not covered by any definition are unknown.

In order to represent three-valued theories of this kind, we adopt Extended Logic Programs (ELP for short) under the well-founded semantics with explicit negation *WFSX* (Pereira & Alferes 1992). In (Pereira, Aparício, & Alferes 1993; Alferes & Pereira 1996) it is shown how ELP can be applied to domains where negative information is made symmetric to positive one, e.g., concept hierarchies, reasoning about actions, counterfactuals, diagnosis, and debugging.

In this work, we consider an extension of Inductive Logic Programming (ILP for short) in order to learn ELP under *WFSX*. As in (Inoue & Kudoh 1997; De Raedt & Bruynooghe 1990), we learn a definition for both a positive concept p and its (explicit) negation $\neg p$. Coverage of examples is tested by adopting the *SLX* interpreter for ELP, defined in (Alferes, Damásio, & Pereira 1994; Alferes & Pereira 1996).

When learning both positive and negative concepts, we may have interaction between the two: their definitions may have a non-empty intersection. We must distinguish two types of atoms in the intersection. *Unseen* atoms, i.e., atoms of target predicates not present in the training set, should be assigned an unknown value, while atoms in the training set should be assigned the truth value of the training set to which they belong and be considered as *exceptions* for the opposite definition.

Therefore, exceptions to a positive concept are identified from negative examples, whereas exceptions to a negative concept are identified from positive examples.

Explicit negation is used to represent the opposite concept while default negation is used to ensure consistency and to handle exceptions to general rules. Exceptions can be generalized, in their turn, by learning within a hierarchy of defaults.

Major innovations of the work concern both the application of ILP to the case of ELP, by integrating standard ILP algorithms with a top-down interpreter for ELP under a well-founded semantics and the discussion of various approaches to be adopted in order to deal with consistency and exceptions.

The paper is organized as follows. We first introduce the new ILP framework. Then we discuss how to avoid inconsistencies on unseen atoms through the use of non-deterministic rules and to deal with exceptions through negation as default. A description of the learning algorithm follows together with an example of its behaviour. Finally we discuss related works and we conclude.

Learning in a Three-valued Setting

In real world problems, complete information about the world is impossible to achieve and it is necessary to reason and act on the basis of the available partial information. In situations of incomplete knowledge, it is important to distinguish between what is true, what is false and what is unknown.

Such situation is, for example, the one of an agent that incrementally gathers information from the surrounding world and has to select its own actions on the basis of such knowledge. If the agent learns in a two-valued settings, it can encounter the problems that have been highlighted in (De Raedt & Bruynooghe 1990). When learning in a specific to general way, it will learn a cautious definition for the target concept and it will not be able to distinguish what is false from what is not yet known (see figure 1a). Suppose the target predicate represents the allowed actions, then the agent will not distinguish forbidden actions from actions with an unknown outcome and this can clearly be a limitation. If the agent learns in a general to specific way, instead, it will not know the difference between what is true and what is unknown (figure 1b) and therefore it can try actions with an unknown outcome. Insted, by learning in a three-valued settings, it will be able to distinguish between allowed actions, forbidden actions and actions with an unknown outcome (figure 1c). In this way, the agent will know which part of the domain needs to be further explored and will not try actions with an unknown outcome unless it is trying to expand its knowledge.

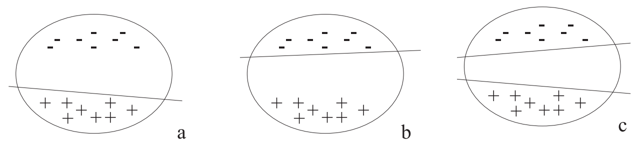

Figure 1: (a,b): two-valued settings, (c): three-valued setting (taken from (De Raedt & Bruynooghe 1990))

Learning in a three-valued settings requires the adoption of a more expressible class of programs to be learned. This class can be represented by means of Extended Logic Programs, under a stable semantics (Gelfond & Lifschitz 1991), or under a well-founded one (Pereira & Alferes 1992). In the following, we will adopt the well-founded semantics with explicit negation *WFSX* (Pereira & Alferes 1992). We will denote negation by default by *not* and explicit negation by $\neg$. $\neg A$ is said the *opposite* literal of A (and viceversa) and *not* A the *complementary* literal of A (and viceversa).

Starting from a set of positive and negative examples for a target predicate p and a background knowledge which is itself an extended logic program under *WFSX*, we apply standard ILP techniques in order to learn a definition for both the positive concept p and its opposite $\neg p$. The ILP learning problem for the case of ELP has been first introduced in (Inoue & Kudoh 1997):

Given:

- a set $\mathcal{P}$ of possible (extended logic) programs
- a set E^+ of positive examples
- a set E^- of negative examples
- a consistent extended logic program B (*background knowledge*)

Find:

- an extended logic program $P \in \mathcal{P}$ such that
  - $B \cup P \models E^+, \neg E^-$ (*completeness*)
  - $B \cup P \not\models E^-, \neg E^+$ (*consistency*)

  where $\neg E = \{\neg e | e \in E\}$, and $E^+, \neg E^-$ (resp. $E^-, \neg E^+$) stands for the conjunction of each atom in E^+ and $\neg E^-$ (resp. in E^- and $\neg E^+$).

Note that, in the ILP problem, it is required that the program is consistent only with respect to the examples. We enlarge this condition requiring that the program is consistent also for unseen atoms, i.e., $B \cup P \not\models L, \neg L$ for every atom L of the target concept.

Since the SLX procedure is correct (in the sense specified in (Alferes & Pereira 1996)), coverage of examples is tested by adopting the SLX top-down interpreter for extended logic programs, defined in (Alferes, Damásio, & Pereira 1994; Alferes & Pereira 1996). It is important to note that neither answer-sets, nor three-valued strong negation, enjoy relevance (Alferes & Pereira 1996; Alferes, Przymusinski, & Pereira 1998), i.e., they cannot have top-down querying procedures, and that is why we use SLX, for $WFSX$ is relevant. We say that an example e is *covered* by program P if $P \vdash e$ according to the SLX procedure.

Therefore the conditions that a program P must satisfy in order to be a solution to the ILP problem can be expressed as "P must cover all the positive examples and all explicit negations of the negative ones" and "P must not cover both a literal and its explicit negation". A theory that satisfies the first condition is said to be *complete*, while a theory that satisfies the second is said to be *consistent*.

The set $\mathcal{P}$ is called the *hypothesis space*. The importance of this set lies in the fact that it defines the search space of the ILP system. In order to be able to effectively learn a program, this space must be restricted as much as possible. If the space is too big, the search could result infeasible.

There are two broad categories of ILP learning methods: *bottom-up* methods, that search the space of clauses specific to general, and *top-down* methods, that search the space of clauses general to specific. In bottom-up methods, clauses are generated by starting with a specific clause that covers one or more positive examples and no negative example, and by iteratively generalizing it as much as possible without covering any negative example. In top-down methods clauses are constructed by starting with a general clause that covers all positive and negative examples and by specializing it until it does no longer cover any negative example while still covering at least one positive.

Relative Least General Generalization (RLGG) (Plotkin 1970), Inverse Resolution (Muggleton & Buntine 1992) and Inverse Entailment (Lapointe & Matwin 1992) are examples of bottom-up techniques. GOLEM (Muggleton & Feng 1990) is a system that learns theories bottom-up by using RLGG. GOLEM generates a single clause by randomly picking couples of examples, by computing their RLGG and by choosing the one with the greatest coverage of positive examples. This clause is further generalized by randomly choosing new positive examples and by computing the RLGG of the clause and each of the example. The generalization that covers more examples is chosen and is further generalized until the coverage of the clause stops increasing. Covered examples are removed from E^+ and the procedure is iterated until no uncovered positive example remains.

Top-down systems, instead, share a basic algorithm that is given as follows (adapted from (Lavrač & Džeroski 1994)):

algorithm LearnTopDown(
    inputs : E^+, E^- : training sets,
        B : background theory,
    outputs : H : learned theory)
Initialize $H := \emptyset$
repeat (*Covering loop*)
    GenerateClause($E^+, E^-, B; c$)
    Remove from E^+ the e^+ covered by c
    Add c to H
until $E^+ = \emptyset$ (*Sufficiency stopping criterion*)

procedure GenerateClause(
    inputs : E^+, E^- : training sets,
        B : background theory,
    outputs : c : clause)
Select a predicate p that must be learned
Initialize c to be $p(\overline{X}) \leftarrow .$
repeat (*Specialization loop*)
    Generate all the possible refinements of c
        by adding a literal L to c
    Find the refinement c_{best} that is best
        according to some heuristic function
    Assign $c := c_{best}$
until c does not cover any negative example
return c (*Necessity stopping criterion*)

FOIL (Quinlan 1990) and Progol (Muggleton 1995) are examples of top-down systems.

Our approach to learning ELP consists in applying ordinary ILP techniques to learn definitions of the positive and negative concept. The ILP technique to be used depends on the level of generality that we want to have for the two definitions: we can look for the Least General Solution (LGS for short) or the Most General Solution (MGS for short). LGSs can be found by adopting a bottom-up method while MGSs can be found by adopting a top-down system.

The generality of the solutions should be chosen independently for the two definitions, thus leading to four epistemological cases depending on the combination of solution generality for the positive and negative concept. The choice of the level of generality should be made on the basis of available knowledge on the domain. Two of the criteria that should be taken into account are the damage that can derive from an erroneous classification of an unseen object and the confidence we have in the training set.

When classifying an unseen object as belonging to a concept, we may later discover that the object belongs to the opposite concept. The more we generalize a concept, the higher number of unseen atoms is covered by the definition and the higher is the risk of an erroneous classification. Depending on the of damage that may derive from such a mistake, we may decide to take a cautious or a confident approach. If the possible damage for a concept is high, then we should learn the LGS for that concept, if the possible damage is low then we can generalize the most and learn the MGS. The risk will depend on the use of the learned concepts within other rules, and so distinct generalities may be employed within the same program.

As regards the confidence in the training set, we can learn the MGS for a concept if we are confident that examples for the opposite concept are correct and representative of the concept. In fact, in top-down methods, negative examples are used in order to limit the generality of the solution. Otherwise, if we think that examples for the opposite concept are not reliable, then we should learn the LGS.

In order to illustrate the difference between the various generalizations, consider the following example.

Example 1 *The domain contains two entities* a, b *and the target concept is* $flies$*.*

$$\begin{array}{ll} bird(a). & animal(a). \\ cat(b). & animal(b). \end{array}$$

The training set is:

$E^+ = \{flies(a)\}$
$E^- = \{flies(b)\}$

Below are reported the most general and least general definitions, for both the positive and negative concept, that constitute a solution to the learning problem:

$$\begin{array}{rcl} flies^+_{MGS}(X) & \leftarrow & bird(X). \\ flies^+_{LGS}(X) & \leftarrow & bird(X), animal(X). \\ flies^-_{MGS}(X) & \leftarrow & cat(X). \\ flies^-_{LGS}(X) & \leftarrow & cat(X), animal(X). \end{array}$$

Intersection of Positive and Negative Definitions

The definitions of the positive and negative concepts may overlap. In this case, we have a contradictory classification for the atoms in the intersection. We propose a representation of the target theory that resolves the conflict by distinguishing two types of atoms in the intersection: those that belong to the training set and those that don't, also called *unseen* atoms (see figure 2).

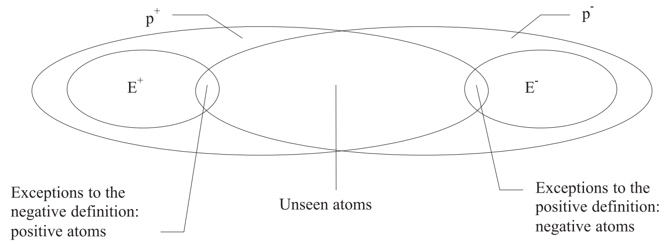

Figure 2: Interaction of the positive and negative definitions on exceptions.

For unseen atoms, the conflict should be resolved by classifying the atoms as unknown, since the arguments for both classifications are equally strong. Instead, for atoms in the training set, the conflict should be resolved by assuming the classification given by the training set, supposing this information is reliable, i.e., in the hypothesis of absence of noise. In other words, atoms in the training set that are covered by the opposite definition should be considered as *exceptions* of that definition.

For unseen atoms in the intersection, the unknown classification is obtained by making the rules non deterministic (Pereira, Aparício, & Alferes 1991; Baral & Gelfond 1994; Alferes & Pereira 1996). The target theory is thus expressed in the following way:

$$\begin{array}{rcl} p(\vec{X}) & \leftarrow & p^+(\vec{X}), not\ \neg p(\vec{X}) \\ \neg p(\vec{X}) & \leftarrow & p^-(\vec{X}), not\ p(\vec{X}) \end{array}$$

where $p^+(\vec{X})$ and $p^-(\vec{X})$ are, respectively, the definitions learned for the positive and the negative concept. For each atom in the intersection, there are two partial stable models, one containing the atom in its positive version, the other containing the opposite literal. The atom is unknown, according to the well-founded semantics for explicit negation (Pereira & Alferes 1992), i.e., in the least partial stable model.

Note that the program $B \cup P$ can be non-stratified, either because the original background is already non-stratified or because the learned program is non-stratified. In this case, three-valued semantics can produce literals with the value "unknown" and one or both of p^+ and p^- may be unknown. If one is unknown and the other is true, then the rules above make both p and $\neg p$ undefined, since the negation by default of an undefined atom is undefined. However, this is counter-intuitive: the defined value should prevail.

In order to handle this case, we suppose that a system predicate $undefined(X)$ is available that succeeds if and only if the atom X is undefined. So we add the following two rules to the definitions for p and $\neg p$:

$$\begin{array}{rcl} p(\vec{X}) & \leftarrow & p^+(\vec{X}), undefined(\neg p(\vec{X})) \\ \neg p(\vec{X}) & \leftarrow & p^-(\vec{X}), undefined(p(\vec{X})) \end{array}$$

According to these clauses, p is true when p^+ is true and $\neg p$ is undefined.

Let us consider now the case in which some atoms in the intersection belong to the opposite training set. We want to classify these atoms according to the classification given by the training set. To this purpose, we add a non-abnormality literal (using negation as default) of the kind $not\ ab_p(\vec{X})$ ($not\ ab_{\neg p}(\vec{X})$) to the rule for p ($\neg p$), expressing the default condition. Then, for every exception $p(\vec{t})$, an individual fact of the form $ab_p(\vec{t})$ ($ab_{\neg p}(\vec{t})$) is asserted (and possibly generalized) so that the rule for p ($\neg p$) will not cover the exception. In this way, exceptions will be present in the model of the theory with the correct definition. The rules thus take the following form:

$$\begin{array}{rcl} p(\vec{X}) & \leftarrow & p^+(\vec{X}), not\ ab_p(\vec{X}), not\ \neg p(\vec{X}) \\ \neg p(\vec{X}) & \leftarrow & p^-(\vec{X}), not\ ab_{\neg p}(\vec{X}), not\ p(\vec{X}) \\ p(\vec{X}) & \leftarrow & p^+(\vec{X}), undefined(\neg p(\vec{X})) \\ \neg p(\vec{X}) & \leftarrow & p^-(\vec{X}), undefined(p(\vec{X})) \end{array}$$

Abnormality literals have not been added to the rules for the undefined case because an atom that is an exception is also an example and it must be covered by the respective definition, therefore it can not be undefined.

In this way, we are able to deal both with the case in which the two definitions are inconsistent and with the case in which exceptions to rules exist, as it is shown in the next example.

Example 2 *Consider a domain containing entities* a, b, c, d, e, f *and suppose the target concept is* fly. *Let the background knowledge be:*

$$\begin{array}{lll} bird(a). & has\_wings(a). & \\ jet(b). & has\_wings(b). & \\ angel(c) & has\_wings(c). & has\_limbs(c). \\ penguin(d) & has\_wings(d). & has\_limbs(d). \\ dog(e). & has\_limbs(e). & \\ cat(f). & has\_limbs(f). & \end{array}$$

and let the training set be:

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

The learned theory is:

$$\begin{array}{rcl} flies(X) & \leftarrow & flies^+(X), not\ ab_{flies}(X), \\ & & not\ \neg flies(X). \\ \neg flies(X) & \leftarrow & flies^-(X), not\ flies(X). \\ flies(X) & \leftarrow & flies^+(X), undefined(\neg flies(X)). \\ \neg flies(X) & \leftarrow & flies^-(X), undefined(flies(X)). \\ ab_{flies^+}(d) & \leftarrow & . \end{array}$$

where

$$\begin{array}{rcl} flies^+(X) & \leftarrow & has\_wings(X). \\ flies^-(X) & \leftarrow & has\_limbs(X). \end{array}$$

Moreover, the abnormality fact $ab_{flies}(d)$ *can be generalized obtaining*
$ab_{flies}(X) \leftarrow penguin(X)$.

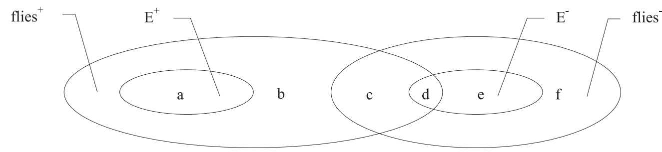

Figure 3: Coverage of definitions for the positive and negative concept

This example (represented in figure 3) shows all the various cases for an atom when learning in a three-valued setting. a and e are examples that are consistently covered by the definitions. b and f are unseen atoms on which there is no contradiction. c and d are atoms where there is contradiction, but c is classified as unknown whereas d is considered as an exception to the positive definition and is classified as negative.

The probability and type of interaction depend on the level of generality of the definitions learned for p^+ and p^-. The higher is the generality, the higher is the probability of interaction between the two.

Algorithm

The algorithm that follows learns ELP of the form described in the previous section:

algorithm LearnELP(
  inputs : E^+, E^- : training sets,
    B : background theory,
  outputs : H : learned theory)
LearnHierarchy($E^+, E^-, B; H_p$)
LearnHierarchy($E^-, E^+, B; H_{\neg p}$)
Obtain H by transforming H_p and $H_{\neg p}$ into
  non-deterministic rules and by adding
  the clauses for the undefined case
output H

procedure LearnHierarchy(
  inputs : E^+ : positive examples,

E^- : negative examples,
B : background theory,
outputs : H : learned theory)
Learn$(E^+, E^-, B; H_p)$
$H := H_p$
for each rule r in H_p **do**
Find the sets E_r^+, E_r^- of positive and negative examples covered by r
if E_r^- is not empty **then**
Add the literal $not\_ab_r(\vec{X})$ to r
Obtain $E_{ab_r}^+$ and $E_{ab_r}^-$ from E_r^- and E_r^+ by transforming each atom $p(\vec{t})$ into $ab_r(\vec{t})$
LearnHierarchy$(E_{ab_r}^+, E_{ab_r}^-, B; H_r)$
$H := H \cup H_r$
endif
enfor
output H

The algorithm uses a procedure LearnHierarchy that, given a set of positive, a set of negative examples and a background knowledge, returns a definition for the positive concept, consisting of default rules, together with definitions for the abnormality literals. The procedure LearnHierarchy is called twice, once for the positive concept and once for the negative concept. In the call for the negative concept, E^- is used as the positive training set and E^+ as the negative one.

LearnHierarchy first calls a procedure Learn$(E^+, E^-, B; H_p)$ that learns a definition H_p for the target concept p. Learn consists of an ordinary ILP algorithm, either bottom-up or top-down, modified to adopt the *SLX* interpreter for testing the coverage of examples and to relax the consistency requirement of the solution. The algorithm thus returns a theory that may cover some negative examples. These negative examples are then treated as exceptions, by adding a default literal to the inconsistent rules and learning a definition for the abnormality predicate. In particular, for each rule $r = p(\vec{X}) \leftarrow Body(\vec{X})$ in H_p, a new non-abnormality literal $not\_ab_r(\vec{X})$ is added to r and a definition for $ab_r(\vec{X})$ is learned by recursively calling LearnHierarchy. Examples for ab_r are obtained from examples for p by observing that, in order to cover a positive (uncover a negative) example $p(\vec{t})$ for p, the atom $ab_r(\vec{t})$ must be false (true). Therefore, positive (negative) examples for ab_r are obtained from the set E_r^- of negative (E_r^+ of positive) examples covered by the rule.

When learning a definition for ab_r, in turn, LearnHierarchy may find exceptions to exceptions and call itself recursively again. In this way we are able to learn a hierarchy of exceptions.

Let us now discuss in more detail the algorithm that implements the Learn procedure. We need an algorithm that, if a consistent solution can not be found, returns a theory that covers the least number of negative examples.

Two approaches are possible. The first consists in learning the least general clause from positive examples only: since the clause is not tested on negative examples, it may cover some of them. This approach can be realized by adopting a bottom-up technique such as RLGG, for example by using the system GOLEM that implements it, as in (Inoue & Kudoh 1997).

The second approach consists in learning from positive and negative examples adopting a top-down learning algorithm where consistency of clauses (necessity stopping criterion in the top-down algorithm) is replaced by a weaker requirement. The simplest criterion that can be adopted is to stop specializing the clause when no literal can be added that reduces the coverage of negative examples.

Other criteria can be used that are based on heuristic functions. For example, the algorithm could stop adding literals when the accuracy rises above a certain threshold, where accuracy is defined as the ratio of covered positive examples over the total number of examples covered by the clause. In ILP, various heuristic necessity stopping criteria that relax the consistency requirement have been designed in order to handle noise. In presence of noise, erroneous information about negative examples may have the effect of causing an overspecialization of clauses in order to make them consistent. Instead, by relaxing the consistency requirement, sufficiently general rules may be learned that cover a limited number of negative examples. These heuristic stopping criteria can be useful as well to learn definitions of concepts with exceptions: when a clause should be specialized too much in order to make it consistent, we prefer to transform it into a default rule and consider the covered negative examples as exceptions.

For example, FOIL (Quinlan 1990) uses a stopping criterion that is based on the *encoding length restriction* (Quinlan 1990) which restrict the total length of an induced clause to the number of bits needed to explicitly enumerate the training examples it covers. The construction of a clause is stopped when adding any literal would cause the length of the clause to exceed the number of bits required to encode the set of examples.

In order to show the behaviour of the algorithm when learning exceptions and to compare it with those of LELP, we will consider the learning problem that is described in example 3.4 in (Inoue & Kudoh 1997) where the definition of the concept $flies$ is learned.

Example 3 *Consider the following background knowledge and training sets:*

$penguin(1)$. $penguin(2)$.
$bird(3)$. $bird(4)$. $bird(5)$.
$bird(X) \leftarrow pen(X)$.
$animal(6)$. $animal(7)$. $animal(8)$.
$animal(9)$. $animal(10)$. $animal(11)$.
$animal(12)$.
$animal(X) \leftarrow bird(X)$.
$E^+ = \{flies(3), flies(4), flies(5)\}$
$E^- = \{flies(1), flies(2), flies(6), flies(7), flies(8)$
$flies(9), flies(10), flies(11), flies(12)\}$

We consider the case in which a top-down method is adopted for the procedure Learn. The stopping criterion used is the simplest, i.e., we stop when no literal can be added to reduce the number of covered negative examples.

The algorithm first learns the positive concept. The first call of the Learn procedure produces the rule
(1) $flies(X) \leftarrow bird(X)$
that is inconsistent since it covers the negative examples $E_1^- = \{flies(1), flies(2)\}$. *Therefore, the rule is specialized by adding a default literal*
(2) $flies(X) \leftarrow bird(X), not\ ab_2(X)$
and LearnHierarchy is called recursively with training sets
$E^+ = \{ab_2(1), ab_2(2)\}$
$E^- = \{ab_2(3), ab_2(4), ab_2(5)\}$
The new call of Learn now returns the rule
(3) $ab_2(X) \leftarrow penguin(X)$
Since the clause is consistent, both recursive calls of LearnHierarchy return and the algorithm starts learning the negative concept. It first generates the rule
(4) $\neg flies(X) \leftarrow animal(X)$
that covers as well the negative examples $E_4^- = \{flies(3), flies(4), flies(5)\}$. *The rule is then transformed into the default rule*
(5) $\neg flies(X) \leftarrow animal(X), not\ ab_5(X)$
and LearnHierarchy is called with training sets
$E^+ = \{ab_5(3), ab_5(4), ab_5(5)\}$
$E^- = \{ab_5(1), ab_5(2), ab_5(6), ab_5(7), ab_5(8),$
$ab_5(9), ab_5(10), ab_5(11), ab_5(12)\}$
Now the following rule is learned
(6) $ab_5(X) \leftarrow bird(X)$
The rule covers the negative examples $E_6^- = \{ab_5(1), ab_5(2)\}$ *and is thus transformed into*
(7) $ab_5(X) \leftarrow bird(X), not\ ab_7(X)$
Finally, LearnHierarchy is called for learning a definition for ab_7 *with the following training sets*
$E^+ = \{ab_7(1), ab_7(2)\}$
$E^- = \{ab_7(3), ab_7(4), ab_7(5),\}$
From this training set, the consistent rule
(8) $ab_7(X) \leftarrow pen(X)$
is generated. The algorithm now terminates by making the clauses for $flies$ *and* $\neg flies$ *non-deterministic and by adding the clauses for the undefined case.*

Related Work

The problems raised by negation and uncertainty in concept-learning, and Inductive Logic Programming in particular, were pointed out in some previous work (e.g., (Bain & Muggleton 1992; De Raedt & Bruynooghe 1990)). For concept learning, the use of the CWA for target predicates is no longer acceptable because it does not allow to distinguish between what is false and what is undefined. To avoid this problem, De Raedt and Bruynooghe (De Raedt & Bruynooghe 1990) proposed to use a three valued logic and an explicit definition of the negated concept in concept learning. This technique has been integrated within the CLINT system, an interactive concept-learner. In the resulting system, both a positive and a negative definition are learned for a concept (predicate) p, stating, respectively, the conditions under which p is true and false. Furthermore, it is required that the concept descriptions be consistent.

The system LELP (Learning ELP) (Inoue & Kudoh 1997) learns ELP under answer-set semantics. As our algorithm, LELP is able to learn non-deterministic default rules with a hierarchy of exceptions. From the point of view of the learning problems that the two algorithms can solve, they are equivalent when the background is a definite logic program: all the examples shown in (Inoue & Kudoh 1997) can be learned by our algorithm and, viceversa, example 2 can be learned by LELP.

When the background is an ELP, instead, the adoption of a well-founded semantics gives a number of advantages with respect to the answer-set semantics. For non-stratified background theories, answer-sets semantics does not enjoy the structural property of relevance (Dix 1995a; 1995b), like our *WFSX* does, and so they cannot employ top-down proof procedures. For the well-founded semantics, instead, the top-down *SLX* interpreter is available, that can be used for testing the coverage of examples in the learning algorithm. Moreover, by means of *WFSX*, we have introduced a method to choose one concept when the other is undefined which they cannot replicate because in the answer-set semantics one has to compute eventually all answer-sets to find out if an atom is unknown. Last but not least, answer-sets semantics is not cumulative, which implies that you cannot assert what you learn without the risk of changing the semantics of what you learned.

The structure of the two algorithms is similar: LELP first generates candidate rules from a concept using an ordinary ILP framework. Then exceptions are identified (as covered examples of the opposite set) and rules specialized through negation as default and abnormality atoms, which are then assumed to prevent the coverage of exceptions. These assumptions can be, in their turn, generalized to generate hierarchical default rules.

A difference between us and Inoue & Kudoh is in the level of generality of the definitions they can learn. LELP generate clauses from positive examples only therefore it can only employ a bottom-up ILP technique and learn the LGS. Instead, we can choose whether to adopt a bottom-up or a top-down algorithm and we can learn theories of different generality for different target concepts.

Another difference consists in the fact that LELP learns a definition only for the concept that has the highest number of examples in the training set. It learns both positive and negative concepts only when the number of positive examples is close to that of negative ones, while we always learn both concepts.

LELP also differs from our approach because it adds to the theory a clause for the negative concept given in terms of the abnormality literals for the positive concept. For example, in the case of example 2, LELP would produce the following theory:

(9) $flies(X) \leftarrow has\_wings(X), not\ ab_1(X).$

(10) $ab_1(X) \leftarrow penguin(X).$

(11) $\neg flies(X) \leftarrow has\_limbs(X).$

(12) $\neg flies(X) \leftarrow ab_1(X).$

We do not generate clause (12) since, when learning a definition for both $flies$ and $\neg flies$, the examples it covers are already covered by clause (11) and therefore such a clause is redundant.

Several other authors have also addressed the task of learning rules with exceptions (Dimopoulos & Kakas 1995; De Raedt & Bruynooghe 1990). In these frameworks, nonmonotonicity and exceptions are dealt with by learning logic programs with negation. In (Dimopoulos & Kakas 1995) the authors rely on a language which uses a limited form of "classical" (or, better, syntactic) negation together with a priority relation among the sentences of the program (Kakas, Mancarella, & Dung 1994) which can be easily mapped into negation as default.

It is worth mentioning that the treatment of exceptions by means of the addition of a non-abnormality literal to each rule (as we and (Inoue & Kudoh 1997) do) is similar to the approach for declarative debugging followed in (Pereira, Damásio, & Alferes 1993). In order to debug a logic program, in (Pereira, Damásio, & Alferes 1993) the authors first transform it by adding a different default literal to each rule. These literals are then used as assumptions of the correctness of the rule, to be possibly revised in the face of a wrong solution. The debugging algorithm determines the assumptions that led to the wrong solution, thus identifying the incorrect rules.

Non-abnormality literals can also be viewed as new abducible predicates, as done for instance in (Esposito *et al.* 1996; 1998; Inoue 1998). In particular, in (Esposito *et al.* 1996; 1998) the authors have considered the integration and cooperation of induction and abduction in order to learn Abductive Logic Programs (ALP) from (possibly) incomplete background knowledge expressed as ALP in its turn. In order to make a rule for a target predicate p consistent, the rule is specialized by adding a new abducible literal $not\_ab_i(\vec{X})$. Exceptions are ruled out by abducing $ab_i(\vec{t})$ for them. These assumptions are then used to learn a definition for ab_i that describes the class of exceptions. In this way, they are able to learn hierarchies of exceptions.

Conclusions

We have shown that Extended Logic Programs are an appropriate representation formalism for learning in a three-valued setting. In this case, one has to learn a definition for both the target and the opposite concept, by considering positive and negative examples as instances of two disjoint classes.

With ELP we are able to represent opposite concepts and to deal with their interaction. Inconsistencies are dealt with differently according to the type of atom causing them. Unseen atoms are classified as unknown through the adoption of non-deterministic rules, while exceptions are dealt with by non-abnormality defaults.

We have presented an algorithm that learns ELP of the form above and to learn hierarchies of exceptions. The algorithm incorporates a standard ILP algorithm suitably extended to adopt the SLX interpreter for ELP and to relax the consistency requirement. Depending on the type of ILP algorithm, either bottom-up or top-down, we can learn the most general solution or the least general solution for the concept and its opposite. The generality of solutions should be chosen independently for the two concepts on the basis of the damage that can derive from an erroneous classification and of the confidence in the training set.

Acknowledgments

This research was partially funded by the PRAXIS XXI project MENTAL, and a NATO sabbatical scholarship to L. M. Pereira.

References

Alferes, J. J., and Pereira, L. M. 1996. *Reasoning with Logic Programming*, volume 1111 of *LNAI*. Heidelberg: SV.

Alferes, J.; Damásio, C.; and Pereira, L. 1994. Top-down query evaluation for well-founded semantics with explicit negation. In *Proceedings of the European Conference on Artificial Intelligenece ECAI94*, 140–144. Morgan Kaufmann.

Alferes, J.; Przymusinski, T.; and Pereira, L. 1998. "Classical" negation in non-monotonic reasoning and logic programming. *Journal of Automated Reasoning* 1.

Bain, M., and Muggleton, S. 1992. Non-monotonic learning. In Muggleton, S., ed., *Inductive Logic Programming.* Academic Press. 145–161.

Baral, C., and Gelfond, M. 1994. Logic programming and knowledge representation. *Journal of Logic Programming* 19/20:73–148.

De Raedt, L., and Bruynooghe, M. 1990. On negation and three-valued logic in interactive concept learning. In *Proceedings of the 9th European Conference on Artificial Intelligence.*

Dimopoulos, Y., and Kakas, A. 1995. Learning Nonmonotonic Logic Programs: Learning Exceptions. In *Proceedings of the 8th European Conference on Machine Learning.*

Dix, J. 1995a. A classification-theory of semantics of normal logic programs: I. strong properties. *Fundamenta Informaticae* XXII(3):227–255.

Dix, J. 1995b. A classification-theory of semantics of normal logic programs: Ii. weak properties. *Fundamenta Informaticae* XXII(3):257–288.

Esposito, F.; Lamma, E.; Malerba, D.; Mello, P.; Milano, M.; Riguzzi, F.; and Semeraro, G. 1996. Learning abductive logic programs. In Denecker, M.; De Raedt, L.; Flach, P.; and Kakas, A., eds., *Proceedings of the ECAI96 Workshop on Abductive and Inductive Reasoning.* Catholic University of Leuven.

Esposito, F.; Ferilli, S.; Lamma, E.; Mello, P.; Milano, M.; Riguzzi, F.; and Semeraro, G. 1998. Cooperation of abduction and induction in logic programming. In Flach, P., and Kakas, A., eds., *Abductive and Inductive Reasoning*, Pure and Applied Logic. Kluwer. Submitted for publication.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programming and disjunctive databases. *New Generation Computing* 9:365–385.

Inoue, K., and Kudoh, Y. 1997. Learning extended logic programs. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, 176–181. Morgan Kaufmann.

Inoue, K. 1998. Learning abductive and nonmonotonic logic programs. In Flach, P., and Kakas, A., eds., *Abductive and Inductive Reasoning*, Pure and Applied Logic. Kluwer. Submitted for publication.

Kakas, A.; Mancarella, P.; and Dung, P. 1994. The acceptability semantics for logic programs. In *Proceedings of the 11th International Conference on Logic Programming.*

Lapointe, S., and Matwin, S. 1992. Sub-unification: A tool for efficient induction of recursive programs. In Sleeman, D., and Edwards, P., eds., *Proceedings of the 9th International Workshop on Machine Learning*, 273–281. Morgan Kaufmann.

Lavrač, N., and Džeroski, S. 1994. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood.

Muggleton, S., and Buntine, W. 1992. Machine invention of first-order predicates by inverting resolution. In Muggleton, S., ed., *Inductive Logic Programming.* Academic Press. 261–280.

Muggleton, S., and Feng, C. 1990. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, 368–381. Ohmsma, Tokyo, Japan.

Muggleton, S. 1995. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming* 13(3-4):245–286.

Pereira, L., and Alferes, J. 1992. Well founded semantics for logic programs with explicit negation. In *Proceedings of the European Conference on Artificial Intelligenece ECAI92*, 102–106. John Wiley and Sons.

Pereira, L.; Aparício; and Alferes, J. 1991. Nonmonotonic reasoning with well founded semantics. In *Proceedings of the International Conference on Logic Programming ICLP91*, 475–489. The MIT Press.

Pereira, L.; Aparício; and Alferes, J. 1993. Non-monotonic reasoning with logic programming. *Journal of Logic Programming* 17:227–263.

Pereira, L. M.; Damásio, C. V.; and Alferes, J. J. 1993. Diagnosis and debugging as contradiction removal. In Pereira, L. M., and Nerode, A., eds., *Proceedings of the 2nd International Workshop on Logic Programming and Non-monotonic Reasoning*, 316–330. MIT Press.

Plotkin, G. 1970. A note on inductive generalization. In *Machine Intelligence*, volume 5. Edinburgh University Press. 153–163.

Quinlan, J. 1990. Learning logical definitions from relations. *Machine Learning* 5:239–266.

Reiter, R. 1978. On closed-word data bases. In Gallaire, H., and Minker, J., eds., *Logic and Data Bases.* Plenum Press. 55–76.