Strategies for 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 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.

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

Standard Inductive Logic Programming techniques are used to learn the concept and its opposite. Depending on the adopted technique, we can learn the most general or the least general definition. Thus, four epistemological varieties occur, resulting from the combination of most general and least general solutions for the positive and negative concept. We discuss the factors that should be taken into account when choosing and strategically combining the generality levels.

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 false, on the basis of the Closed World Assumption (CWA) (Reiter 1978). However in practice, it is more often the case that we are confident about the truth or falsity of only a limited number of facts, and are not able to draw any conclusion about the remaining ones, because the available information 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. Such an agent needs to distinguish between what is true, what is false and what is unknown, and therefore needs to learn within a three-valued setting.

For this purpose, we adopt the class of *Extended Logic Programs* (ELP, for short, in the sequel) as the representation language for learning. ELP are logic programs with default negation, plus a second form of negation, called *explicit*, which have been recognized as a very expressive means of representation. The adoption of ELP as a representation formalism allows one to deal directly with incomplete knowledge, by expressing *unknown* information through default negation as well as *negative* information through explicit negation (Alferes & Pereira 1996). For instance, in (Pereira, Aparício, & Alferes 1993; Alferes & Pereira 1996) it is shown how ELP are applicable to such diverse domains of knowledge representation as concept hierarchies, reasoning about actions, counterfactuals, diagnosis, and debugging.

In this work, we discuss various approaches and strategies that can be adopted in Inductive Logic Programming (ILP, henceforth) for learning ELP. As in (Inoue & Kudoh 1997; Inoue 1998), the learning process starts from a set of positive and negative examples plus some background knowledge in the form of a logic program extended with explicit negation. Positive and negative information in the training set are treated equally, by learning a definition for both a positive concept p and its (explicitly) negated concept $\neg p$. Coverage of examples is tested by adopting the SLX interpreter for extended logic programs under the well founded semantics, defined in (Alferes, Damásio, & Pereira 1994; Alferes & Pereira 1996).

Default negation is used in the learning process 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. A definition for the class of exceptions is then learned which may include new exceptions. The process is then iterated thus possibly producing a hierarchy of exceptions.

Furthermore, in order to handle *contradiction*, and avoid it whenever possible, learned rules are then made non-deterministic (see (Pereira, Aparício, & Alferes 1991; Baral & Gelfond 1994)) by making the learned definition for a positive concept p depend on the default negation of the negative concept $\neg p$, and vice-versa.

In this work, we adopt standard ILP techniques to learn some concept and its opposite. Depending on the technique adopted, one can learn the most general or the least general definition for each concept. Thus, four epistemological varieties occur, resulting from the mutual combination of most general and least general solutions for the positive and negative concept. We discuss some of the factors that should be taken into account when choosing the level of generality of each, and their combination, to define a specific learning strategy.

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 algorithm follows together with an example of its behaviour We then discuss the factors that should be taken into account when choosing the level of generality of the learned theories. 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. Instead, 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.

Learning in a three-valued settings requires the adoption of a more expressive class of programs to be learned. This class can be represented by means of Extended Logic Programs, either under a stable semantics (Gelfond & Lifschitz 1991) or under a well-founded one (Pereira & Alferes 1992). An extended logic program is a set of rules of the form

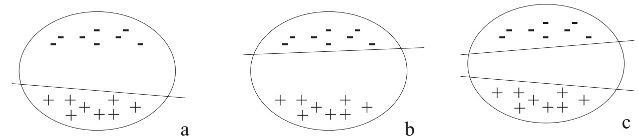

Figure 1: (a,b): two-valued settings, (c): three-valued setting (taken from (De Raedt & Bruynooghe 1990))

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_{m+n}$$

where $m, n \geq 0$ and each l_i $(i = 1, \ldots, m+n)$ is an *objective* literal, i.e., a literal of the form a or $\neg a$, where a is an atom. Thus, from now on we will distinguish between default (or *implicit*) negation, denoted by *not*, and a second form of (*explicit*) negation, denoted by $\neg$. $\neg A$ is said the *opposite* literal of A (and viceversa) and *not* A the *complementary* literal of A (and viceversa). The coherence principle (Pereira & Alferes 1992) relates the two negations by stipulating that $\neg L$ entails *not* L for any objective literal L.

In the following, we will adopt the well-founded semantics with explicit negation *WFSX* (Pereira & Alferes 1992) because none of the other semantics for ELP, answer-sets and 3-valued strong negation, enjoys the property of relevance (Alferes & Pereira 1996; Alferes, Przymusinski, & Pereira 1998), i.e., they cannot have top-down querying procedures. Instead, for *WFSX* there exists the top-down proof procedure SLX (Alferes, Damásio, & Pereira 1994; Alferes & Pereira 1996), which is correct with respect to the semantics as per (Alferes & Pereira 1996).

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 a positive concept p and its opposite $\neg p$. The ILP learning problem for the case of ELP was first introduced in (Inoue & Kudoh 1997), and is defined here as follows.

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 e in E^+ and in $\neg E^-$ (resp. in E^- and $\neg E^+$).

Examples are ground atoms of target predicates and the theory that is learned will contain rules of the form $p(\vec{X}) \leftarrow Body^{+}(\vec{X})$ and $\neg p(\vec{X}) \leftarrow Body^{-}(\vec{X})$ for every target predicate p, where $\vec{X}$ stands for a tuple of arguments. In order to satisfy the completeness requirement, the rules for p will entail all positive examples while the rules for $\neg p$ will entail all (negated) negative examples. The consistency requirement is satisfied by ensuring that both sets of rules do not entail examples of the opposite training set.

Note that, in the case of ELP, the consistency with respect to the training set is equivalent to the requirement that the program is non-contradictory on the examples. This requirement is enlarged to require 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 predicates.

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. We say that an example e is *covered* by program P if $P \vdash e$ according to the SLX procedure.

Basically, the SLX derivation procedure relies on two kinds of derivations: T-derivations, proving truth, and TU-derivations proving non-falsity. Shifting from one to the other is required for proving a default literal *not* L: the T-derivation of *not* L succeeds if and only if the TU-derivation of L fails; the TU-derivation of *not* L succeeds if and only if the T-derivation of L fails. Moreover, T-derivation of *not* L succeeds if T-derivation of $\neg L$ succeeds (thus taking into account the *coherence principle*), and TU-derivation of L fails if T-derivation of $\neg L$ succeeds.

The SLX procedure is amenable to a simple implementation. To guarantee termination, suitable rules are introduced that prune the search space and eliminate both cyclic positive recursion and cyclic negative recursion.

Our approach to learning ELP consists in applying conventional ILP techniques to learn the positive and the negative definition where the SLX procedure substitutes the Prolog procedure to test the coverage of examples.

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) of the problem of learning each concept. In practice, LGS and MGS are not unique and real systems usually learn theories that are not the least or most general but closely approximate one of the two. In the following, these concepts will be used to mean approximations of the theoretical concepts.

LGSs can be found by adopting a bottom-up method such as Relative Least General Generalization (RLGG) (Plotkin 1970), Inverse Resolution (Muggleton & Buntine 1992) or Inverse Entailment (Lapointe & Matwin 1992). Instead, MGSs can be found by adopting a top-down system such as FOIL (Quinlan 1990) or Progol (Muggleton 1995).

The generality of the solutions should be chosen independently for the two definitions, thus leading to four epistemological cases, that will be discussed in a following section.

The problem of learning a concept and its opposite in a three-valued settings can be generalized to the problem of learning the definitions of n disjoint classes. In the following section we will show that ELP can be used successfully also to represent the solutions to this generalized learning problem.

Eliminating Learned Contradictions

The definitions of the positive and negative concepts may overlap. In this case, we have a contradictory classification for the atoms in the intersection. In order to resolve the conflict, we must distinguish 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).

Theories are tested for consistency on all the atoms of the training set, so we should not have a conflict on them. However, in some cases, it is useful to relax the consistency requirement and to learn clauses that cover a small amount of negative examples. This happens when it would be otherwise impossible to learn a definition for the concept, because no clause in the language bias is consistent, or when an overspecific definition would be learned, composed of very many specific clauses instead of a few general ones. In these cases, the definitions of the positive and negative concepts may cover atoms of the opposite training set. These atoms must then be considered as exceptions.

Exceptions may be due to noise in the collection of data or to abnormalities in the opposite concept. In the second case, if exceptions form a class, it may be possible to learn a definition for it, provided that we have data on their common properties and the language bias allows such a definition.

In the following we discuss how to resolve the conflict in the case of atoms in the training set and of unseen atoms. We first consider the case in which the training sets are disjoint and we later extend the discussion to the case where there is a non-empty intersection of the training sets.

For unseen atoms, the conflict is resolved by classifying the atoms as unknown, since the arguments supporting the two classifications are equally strong. Instead, for atoms in the training set, the conflict is resolved by giving priority to the classification stipulated by the training set. In other words, atoms in a training set that are covered by the opposite definition are considered as *exceptions* to that definition.

For unseen atoms in the intersection, the unknown classification is obtained by making the rules non deterministic (see (Pereira, Aparício, & Alferes 1991;

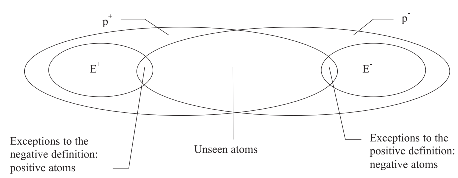

Figure 2: Interaction of the positive and negative definitions on exceptions.

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 $p(\vec{t})$ in the intersection of p^+ and p^-, there are two partial stable models, one containing the atom in its positive version, the other containing the opposite literal. According to the well-founded semantics for explicit negation (Pereira & Alferes 1992) the atom is unknown, i.e., no literal $p(\vec{t})$, $\neg p(\vec{t})$, $not\ p(\vec{t})$ or $not\ \neg p(\vec{t})$ belongs to the least partial stable model. The resulting program contains a recursion through negation (i.e., it is *non stratified*) but the *SLX* procedure does not go into a loop because it has special mechanisms for loop detection and avoidance.

Note that $p^+(\vec{X})$ and $p^-(\vec{X})$ can assume as well the undefined truth value, either because the original background is non-stratified or because they rely on the definition learned for another target predicate, that is of the form above and therefore 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 pertaining to the concept in question, such atoms representing *exceptions*. To handle exceptions to classification rules, we add a negative literal (using negation as default) of the form $not\ abnorm_{p^+}(\vec{X})$ (resp. $not\ abnorm_{p^-}(\vec{X})$) to the rule for p (resp. $\neg p$), expressing the default condition. Then, for every exception $p(\vec{t})$, an individual fact of the form $abnorm_{p^+}(\vec{t})$ (resp. $abnorm_{p^-}(\vec{t})$) is asserted so that the rule for p (resp. $\neg p$) will not cover the exception, while the opposite definition will still cover it. In this way, exceptions will figure in the model of the theory with the correct truth value. The learned theory thus takes the following form:

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

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.

The individual facts of the form $abnorm_{p^+}(\vec{t})$ are then used as examples for learning a definition for $abnorm_{p^+}$ and $abnorm_{p^-}$, as in (Inoue & Kudoh 1997; Lamma *et al.* 1997). In turn, exceptions to the definitions of $abnorm_{p^+}$ and $abnorm_{p^-}$ may be found and so on, thus leading to a hierarchy of exceptions.

We now consider the case in which the training sets are not disjoint. In this case, atoms in the intersection of the training sets are considered as exceptions for both definitions. For an atom $p(\vec{t})$, both $p(\vec{t})$ and $\neg p(\vec{t})$ will result false in the three-valued model of the theory, i.e., $not\ p(\vec{t})$ and $not\ \neg p(\vec{t})$ will belong to the model. Therefore, these atoms differ from unseen ones, for which the truth value of $p(\vec{t})$ and $\neg p(\vec{t})$ is unknown.

Example 1 *Consider a domain containing entities* a, b, c, d, e, f *and suppose the target concept is* $flies$. *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:

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

where

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

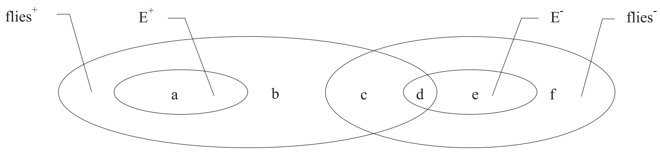

Figure 3: Coverage of definitions for the positive and negative concept

Moreover, the abnormality fact $ab_{flies}(d)$ *can be generalized obtaining*

$ab_{flies}(X) \leftarrow penguin(X).$

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.

As noted in the previous section, ELP can be used as well to represent n disjoint classes $p_1, \ldots, p_n$. When you have to learn n disjoint classes, the training set contains a number of atoms associated with their class. Let p_i^+ be a definition for p_i learned by using, as positive examples, the atoms in the training set classified as belonging to p_i and, as negative examples, all the atoms for the other classes. Then the following rules ensure consistency on unseen atoms and on exceptions:

$$
\begin{array}{lcl}
p_1(\vec{X}) & \leftarrow & p_1^+(\vec{X}), not\ ab_{p_1^+}(\vec{X}), not\ p_2(\vec{X}),\\
 & & \ldots, not\ p_n(\vec{X}).\\
p_2(\vec{X}) & \leftarrow & p_2^+(\vec{X}), not\ ab_{p_2^+}(\vec{X}), not\ p_1(\vec{X}),\\
 & & not\ p_3(\vec{X}), \ldots, not\ p_n(\vec{X}).\\
\cdots & \leftarrow & \cdots\\
p_n(\vec{X}) & \leftarrow & p_n^+(\vec{X}), not\ ab_{p_n^+}(\vec{X}), not\ p_1(\vec{X}),\\
 & & \ldots, not\ p_{n-1}(\vec{X}).
\end{array}
$$

$$
\begin{array}{lcl}
p_1(\vec{X}) & \leftarrow & p_1^+(\vec{X}), undef(p_2(\vec{X})), \ldots, undef(p_n(\vec{X})).\\
p_2(\vec{X}) & \leftarrow & p_2^+(\vec{X}), undef(p_1(\vec{X})),\\
 & & undef(p_3(\vec{X})), \ldots, undef(p_n(\vec{X})).\\
\cdots & \leftarrow & \cdots\\
p_n(\vec{X}) & \leftarrow & p_n^+(\vec{X}), undef(p_1(\vec{X})), \ldots, undef(p_{n-1}(\vec{X})).
\end{array}
$$

regardless of the algorithm that is used for learning p_i^+.

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 eventual 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 details the algorithm that implements the Learn procedure. We need an algorithm that, if a consistent solution cannot 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 (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 over-specialization 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 restricts 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 2 *Consider the following background knowledge and training sets:*

$penguin(1).\quad penguin(2).$
$bird(3).\quad bird(4).\quad bird(5).$
$bird(X) \leftarrow pen(X).$
$animal(6).\quad animal(7).\quad animal(8).$
$animal(9).\quad animal(10).\quad animal(11).$
$animal(12).$
$animal(X) \leftarrow bird(X).$

$E^+ = \{flies(3), flies(4), flies(5)\}$
$E^- = \{flies(1), flies(2), flies(6), flies(7), flies(8)$
$\quad flies(9), flies(10), flies(11), flies(12)\}$

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

$E^+ = \{ab_5(3), ab_5(4), ab_5(5)\}$
$E^- = \{ab_5(1), ab_5(2), ab_5(6), ab_5(7), ab_5(8),$
$\quad ab_5(9), ab_5(10), ab_5(11), ab_5(12)\}$

Now the following rule is learned

(6) $ab_5(X) \leftarrow bird(X).$

This rule, in its turn, 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) $ab_7(X) \leftarrow penguin(X).$

is generated. The algorithm makes the clauses for $flies$ *and* $\neg flies$ *non-deterministic, adds the clauses for the undefined case and then terminates.*

Strategically Combining Different Generalizations

The generality of the solutions to be learned is an important issue when learning in a three-valued setting. In a two-valued setting, once the generality of the definition is chosen, the extension (i.e., the generality) of the set of false atoms is automatically decided, since it is the complement of the true atoms set. Instead, in a three-valued setting, the extension of the set of false atoms depends on the generality of the definition learned for the negative concept. Therefore, the level of generality must be chosen independently for both definitions, thus obtaining four possible combinations.

Moreover, the generality of the solutions learned for the positive and negative concept clearly influences the interaction between the definitions. If we learn the MGS for both the concept and its opposite, the probability that their intersection is non-empty is higher than if we learn the LGS for both definitions. Therefore, the decision on which type of solution to learn should take into account the possibility of interaction as well: if we want to reduce this possibility, we have to learn two LGSs, if we do not care about interaction, we can learn two MGSs. In general, we may learn different generalizations and combine them in distinct ways for different strategic purposes within the same application problem.

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 is the number of unseen atoms 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.

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 the following we present an example of the kind of reasoning that can be used to choose the level of generality, and we discuss how to strategically combine the different levels by employing the extended logic programming approach to learning.

Consider a person living in a bad neighbourhood in Los Angeles. He is an honest man and to survive he needs two concepts, one about who is likely to attack him, on the basis of appearance, band membership, age, past dealings, etc. Since he wants to take a cautious approach, he maximizes $attacker^+$ and minimizes $attacker^-$, so that his $attacker1$ concept allows him to avoid dangerous situations.

$attacker1(X) \leftarrow attacker^+_{MGS}(X), not\ \neg attacker1(X).$

$\neg attacker1(X) \leftarrow attacker^-_{LGS}(X), not\ attacker1(X).$

Another concept he needs is the type of beggars he should give money to (he is a good man) that actually seem to deserve it, on the basis of appearance, health, age, children, etc. Since he is not rich and also he doesn't like to be tricked, he learns a $beggar1$ concept by minimizing $beggar^+$ and maximizing $beggar^-$, so that his beggar concept allows him to give money strictly to those appearing to need it without faking.

$beggar1(X) \leftarrow beggar^+_{LGS}(X), not\ \neg beggar1(X).$

$\neg beggar1(X) \leftarrow beggar^-_{MGS}(X), not\ beggar1(X).$

However rejected beggars, especially malicious ones, may turn into attackers, in this very bad neighbourhood. Consequently, if he thinks a beggar might attack him he'd better be more permissive about what is a beggar and placate him with money. In other words, he should maximize $beggar^+$ and minimize $beggar^-$ in a $beggar2$ concept.

$beggar2(X) \leftarrow beggar^+_{MGS}(X), not\ \neg beggar2(X).$

$\neg beggar2(X) \leftarrow beggar^-_{LGS}(X), not\ beggar2(X).$

These concepts can be used in order to minimize his risk taking when he carries a lot of money, with the following kind of reasoning:

$run \leftarrow lot\_of\_money, attacker1(X), not\ beggar2(X).$

$\neg run \leftarrow give\_money.$

$give\_money \leftarrow beggar1(X).$

$give\_money \leftarrow attacker1(X), beggar2(X).$

If he doesn't have a lot of money with him, he may prefer not to run as he risks being beaten up. In this case he has to relax his attacker concept into $attacker2$, but not relaxed so much that he would use $attacker^-_{MGS}$.

$\neg run \leftarrow little\_money, attacker2(X).$

$attacker2(X) \leftarrow attacker^+_{LGS}(X), not\ \neg attacker2(X).$

$\neg attacker2(X) \leftarrow attacker^-_{LGS}(X), not\ attacker2(X).$

Related Work

The adoption of a three-valued logic in learning have been investigated by many authors. Many propositional learning systems learn a definition for both the concept and its opposite. For example, systems that learn decision trees, as c4.5 (Quinlan 1993), or decision rules, as the AQ family of systems (Michalski 1973), are able to solve the problem of learning a definition for n classes, that generalizes the problem of learning a concept and its opposite. However, in most cases the definitions learned are assumed to cover the whole universe of discourse: no unknown classification is produced, any instance is always classified as belonging to one of the classes. Instead, we classify as unknown the instances for which the learned definitions do not give an unanimous response.

When learning multiple concepts, it may be the case that the descriptions learned are overlapping. We have considered this case as non-desirable: this is reasonable when learning a concept and its opposite but it may not be the case when learning more than two concepts. As it has been pointed out by (Michalski 1984), in some cases, it is useful to produce more than one classification for an instance, for example if a patient has two diseases, his symptoms should satisfy the descriptions of both diseases. Subject for future work will be to consider classes of paraconsistent logic programs where the overlapping of definitions for p and $\neg p$ is allowed.

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. 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 those under which it is false. The definitions are learned so that they do not produce an inconsistency on the examples. Differently from this system, we take also care that the two definitions do not produce inconsistency on unseen atoms and we are able to learn definitions for exceptions to both concepts.

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 1 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 1, 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). 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 (as we and (Inoue & Kudoh 1997) do) to each rule 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 (Lamma *et al.* 1997; Esposito *et al.* 1998; Inoue 1998). In particular, in (Lamma *et al.* 1997; Esposito *et al.* 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 $not\_abnorm_i(\vec{X})$. Exceptions are then ruled out by assuming $abnorm_i(\vec{t})$ for them. These assumptions are then used to learn a definition for $abnorm_i$ 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 different 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 atoms causing them. Unseen atoms are classified as unknown through the adoption of non-deterministic rules, while exceptions are dealt with by non-abnormality defaults.

The algorithm for learning ELP that we propose adopts standard ILP techniques for learning the definitions of the concept and its opposite. ILP algorithms are modified so that they employ the *SLX* procedure for testing the coverage of examples and the consistency requirement is relaxed in order to allow exceptions to rules. The algorithm is then able to learn a definition for the exceptions, possibly containing new exceptions and so on, thus building a hierarchy of exceptions.

Depending on the type of ILP technique adopted, we can learn the Least General Solution or the Most General Solution for the positive and negative concept. The generality of the solutions should be chosen independently for the two definitions, thus leading to four epistemological cases which can be strategically combined to achieve diversely useful learned concepts. Future work will be devoted to identify the type of learning problems more appropriate for each case.

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: Springer-Verlag.

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 Non-monotonic 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.; 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.*

Lamma, E.; Mello, P.; Milano, M.; and Riguzzi, F. 1997. Integrating Induction and Abduction in Logic Programming. In Wang, P. P., ed., *Proceedings of the Third Joint Conference on Information Sciences*, volume 2, 203–206.

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.

Michalski, R. 1973. Discovery classification rules using variable-valued logic system VL1. In *Proceedings of the Third International Conference on Artificial Intelligence*, 162–172. Stanford University.

Michalski, R. 1984. A theory and methodology of inductive learning. In Michalski, R.; Carbonell, J.; and Mitchell, T., eds., *Machine Learning - An Artificial Intelligence Approach*, volume 1. Springer-Verlag. 83–134.

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. 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.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann.

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