Agents Learning in a Three-Valued Logical Setting

Evelina Lamma 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 show that the adoption of a three-valued setting for inductive concept learning is particularly useful for learning in single and multiple agent systems.

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. Such situation occurs, for example, when an agent incrementally gathers information from the surrounding world and has to select its own actions on the basis of such acquired knowledge.

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 (ELP) under a Well-Founded Semantics with explicit negation (*WFSX*) as the representation formalism for learning. Standard Inductive Logic Programming techniques are then employed to learn the concept and its opposite.

The learnt definitions of the positive and negative concepts may overlap, both when learning conflicting rules for a predicate and its explicit negation by a single agent or when combining the knowledge learned by multiple agents. In the paper, we handle the issue of strategic combination of possibly contradictory learnt definitions.

1 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) [25]. 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. Like it has been pointed out in [9, 8], this is typically the case of an autonomous agent that, in an incremental way, 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 richer 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 in a three-valued setting. ELP contain two kinds of negation: default negation plus a second form of negation, called *explicit*, whose combination has been recognized as a very expressive means of knowledge representation. The adoption of ELP allows one to deal directly in the language with incomplete knowledge, with exceptions through default negation, as well as with truly *negative* information through explicit negation [22, 3, 4]. For instance, in [3, 5, 10, 7, 18] it is shown how ELP are applicable to such diverse domains of knowledge representation as concept hierarchies, reasoning about actions, belief revision, counterfactuals, diagnosis, updates and debugging.

In this work, we first discuss various approaches and strategies that can be adopted in Inductive Logic Programming (ILP, henceforth) for learning with a three-valued settings by a single agent. Then, we discuss how the knowledge learned by separate agents can be combined to obtain a common knowledge base.

As in [14, 13], the learning process in a single agent starts from a set of positive and negative examples plus some background knowledge in the form of an extended logic programs. 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 $\neg p$ by means of standard ILP techniques. Coverage of examples is tested by adopting the SLX interpreter for ELP under the Well-Founded Semantics with explicit negation (*WFSX*) defined in [3, 10].

Indeed, separately learned positive and negative concepts may conflict and, in order to handle possible *contradiction*, contradictory learned rules are defused by making the learned definition for a positive concept p depend on the default negation of the negative concept $\neg p$, and vice-versa. I.e., each definition is introduced as an exception to the other. This way of coping with contradiction can be generalized in order to combine knowledge learned by different agents, by taking also into account preferences among multiple learning agents or information sources.

The paper is organized as follows. We first provide some preliminaries on the language of ELP in 2. Then we motivate the use of ELP as target and background language and we introduce the new ILP framework in section 3. Section 4 proposes how to avoid inconsistencies on unseen atoms and their opposites, through the use of mutually defusing ("non-deterministic") rules, for the case of single and multiple learning agents, and how to incorporate exceptions through negation by default. Finally, we examine related works in section 5 and conclude in section 6.

2 Extended Logic Programs

An *extended logic program* is a finite set of rules of the form:

$$L_0 \leftarrow L_1, \ldots, L_n$$

with $n \geq 0$, where L_0 is an objective literal, $L_1, \ldots, L_n$ are literals and each rule stands for the sets of its ground instances. *Objective literals* are of the form A or $\neg A$, where A is an atom, while a *literal* is either an objective literal L or its default negation $not\ L$. $\neg A$ is said the *opposite* literal of A (and vice versa), where $\neg\neg A = A$, and $not\ A$ the *complementary* literal of A (and vice versa). By $not\ \{a_1, \ldots, a_n\}$ we mean $\{not\ a_1, \ldots, not\ a_n\}$ where a_is are literals. By $\neg\ \{a_1, \ldots, a_n\}$ we mean $\{\neg a_1, \ldots, \neg a_n\}$. The set of all objective literals of a program P is called its *extended Herbrand base* and is represented as $H^E(P)$. An *interpretation* I of an extended program P is denoted by $T \cup not\ F$, where T and F are disjoint subsets of $H^E(P)$. Objective literals in T are said to be *true* in I, objective literals in F are said to be *false* in I and those in $H^E(P) - I$ are said to be *undefined* in I. We introduce in the language the proposition $\mathbf{u}$ that is undefined in every interpretation I.

The *WFSX* extends the well founded semantics (*WFS*) [26] for normal logic programs to the case of extended logic programs. *WFSX* is obtained from *WFS* by adding the coherence principle relating the two forms of negation: "*if L is an objective literal and $\neg L$ belongs to the model of a program, then also $not\ L$ belongs to the model*", i.e., $\neg L \rightarrow not\ L$. See [3, 10] for a definition of *WFSX*.

Let us now show an example of the *WFSX* semantics in the case of a simple program.

Example 1 *Consider the following extended logic program:*

$$\begin{array}{lll} \neg a & \leftarrow & . \\ a & \leftarrow & b. \end{array} \qquad\qquad b \leftarrow not\ b.$$

A WFSX *model of this program is $M = \{\neg a, not\ \neg b, not\ a\}$: $\neg a$ is true, a is false (i.e., both $\neg a$ and $not\ a$ are in the well-founded model), $\neg b$ is false (there are no rules for $\neg b$) and b is undefined. Notice that $not\ a$ is in the model since it is implied by $\neg a$ via the coherence principle.*

One of the most important characteristic of *WFSX* is that it provides a semantics for an important class of extended logic programs: the set of *non-stratified* programs, i.e., the set of programs that contain recursion through default negation. An extended logic program is *stratified* if its *dependency graph* does not contain any cycle with an arc labelled with $-$. The *dependency graph* of a program P is a labelled graph with a node for each predicate of P and an arc from a predicate p to a predicate q if q appears in the body of clauses with p in the head. The arc is labelled with $+$ if q appears in an objective literal in the body and with $-$ if it appears in a default literal.

Non-stratified programs are very useful for knowledge representation because the *WFSX* semantics assigns the truth value undefined to the literals involved in the recursive cycle through negation. In section 4 we will employ non stratified programs in order to resolve possible contradictions.

WFSX was chosen among the other semantics for extended logic programs, answer-sets [12] and three-valued strong negation [4], because none of the others enjoy the property of relevance [3, 4] for non-stratified programs, i.e., they cannot have top-down querying procedures for non-stratified programs. Instead, for *WFSX* there exists a top-down proof procedure *SLX* [3], which is correct with respect to the semantics[1]. Cumulativity is also enjoyed by *WFSX*, i.e., if you add a lemma then the semantics does not change (see [3]). This property is important for speeding-up the implementation. By memorizing intermediate lemmas through tabling, the implementation of *SLX* greatly improves. Answer-set semantics, however, is not cumulative for non-stratified programs and thus cannot use tabling.

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

The *SLX* procedure is amenable to a simple Prolog implementation [3] that consists in pre-processing *WFSX* programs and mapping them into *WFS* programs through the T-TU transformation [6]. This transformation is linear and essentially doubles the number of program clauses. To guarantee termination, suitable rules are introduced that prune the search space and eliminate both cyclic positive recursion and cyclic negative recursion. Then, the transformed program can be executed in *XSB*, an efficient logic programming system which implements the *WFS* with tabling, and subsumes Prolog. Tabling in *XSB* consists in memoizing intermediate lemmas, and in properly dealing with non-stratification according to *WFS*. Tabling is important in learning, where computations are often repeated for testing the coverage or otherwise of examples.

[1]Though *WFSX* is not truth-functional (i.e., the truth-value of any formula does not depend only on the truth-value of its subformulas as expressed by the truth table of the logical connectives) any extended logic program under *WFSX* can be transformed into an equivalent program under WFS through the T-TU transformation [6, 3] which is truth-functional. This transformation is used for the implementation.

3 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 or undefined.

Such situation occurs, for example, when an agent incrementally gathers information from the surrounding world and has to select its own actions on the basis of such acquired knowledge. If the agent learns in a two-valued setting, it can encounter the problems that have been highlighted in [9]. 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). Supposing the target predicate represents the allowed actions, then the agent will not distinguish forbidden actions from actions with an unknown outcome and this can restrict the agent acting power. 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. Rather, by learning in a three-valued setting, 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.

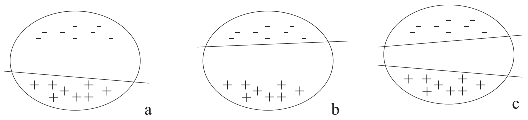

Figure 1: (taken from [9])(a,b): two-valued setting, (c): three-valued setting

Learning in a three-valued setting requires the adoption of a more expressive class of programs to be learned. This class can be represented, we have seen, by means of extended logic programs under the well-founded semantics extended with explicit negation *WFSX* [3, 4, 22].

We therefore consider a new learning problem where we want to learn an extended logic program from a background knowledge that is itself an extended logic program and from a set of positive and a set of negative examples in the form of ground facts for the target predicates. A learning problem for extended logic programs was first introduced in [14] where the notion of coverage was defined by means of truth in the answer-set semantics. Here the problem definition is modified to consider coverage as truth in the *WFSX* semantics

Definition 2 (Learning ELP)

Given:

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

Find:

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

  where $\neg E = \{\neg e | e \in E\}$.

We suppose that the training sets E^+ and E^- are disjoint.

The theory that is learned will contain rules of the following form:

$$p(\vec{X}) \leftarrow Body^+(\vec{X})$$

$$\neg p(\vec{X}) \leftarrow Body^-(\vec{X})$$

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 (explicitly negated) negative examples. The consistency requirement is satisfied by ensuring that both sets of rules do not entail instances of the opposite element in either of the training sets.

Note that, in the case of extended logic programs, 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 be non-contradictory also for unseen atoms, i.e., $B \cup P \not\models L \wedge \neg L$ for every atom L of the target predicates.

We say that an example e is *covered* by program P if $P \models_{WFSX} e$. Since the *SLX* procedure is correct with respect to *WFSX*, even for contradictory programs, coverage of examples is tested by verifying whether $P \vdash_{SLX} e$.

Our approach to learning with extended logic programs consists in initially applying conventional ILP techniques to learn a positive definition from E^+ and E^- and a negative definition from E^- and E^+. In these techniques, the *SLX* procedure substitutes the standard Logic Programming proof procedure to test the coverage of examples.

The ILP techniques to be used depend on the level of generality that we want to have for the two definitions: we can look for the Least General Solution (LGS) or the Most General Solution (MGS) of the problem of learning each concept and its complement. In practice, LGS and MGS are not unique and real systems usually learn theories that are not the least nor most general, but approximate one of the two. In the following, these concepts will be used to signify approximations of the theoretical concepts.

LGSs can be found by adopting one of the bottom-up methods such as relative least general generalization (*rlgg*) [23] and the GOLEM system [21], inverse resolution [20] or inverse entailment [16]. Conversely, MGSs can be found by adopting a top-down refining method (cf. [17]) and a system such as FOIL [24] or Progol [19].

A system has been implemented that is able to solve the above mentioned problem. The system is called LIVE (Learning in a three-Valued Environment) and is described in details in [**?**]. In particular, the system learns a definition for both the concept and its opposite and is parametric in the procedure used for learning each definition: it can adopt either a top-down algorithm, using beam-search and heuristic necessity stopping criterion, or a bottom-up algorithm, that exploits the GOLEM system.

4 Strategies for Eliminating Learned Contradictions

Even for a single agent, the definitions of the positive and negative concepts may overlap. In this case, we have a contradictory classification for the objective literals in the intersection. In order to resolve the conflict, we must distinguish two types of literals in the intersection: those that belong to the training set and those that do not, also dubbed *unseen* atoms (see figure 2).

In the following, we discuss how to resolve the conflict in the case of unseen literals and of literals in the training set. From now onwards, $\vec{X}$ stands for a tuple of arguments.

Contradiction on Unseen Literals For unseen literals, the conflict is resolved by classifying them as undefined, since the arguments supporting the two classifications are equally strong. Instead, for literals in the training set, the conflict is resolved by giving priority to the classification stipulated by the training set. In other words, literals in a training set that are covered by the opposite definition are made as *exceptions* to that definition. For unseen literals in the intersection, the undefined classification is obtained by making opposite rules mutually defeasible, or "non-deterministic" (see [5, 3]). The target theory is consequently expressed in the following way:

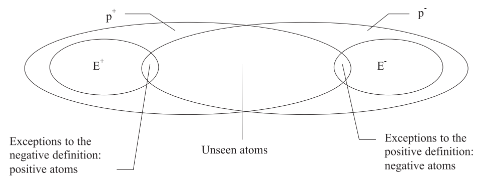

Figure 2: Interaction of the positive and negative definitions on exceptions.

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

where $p^{+}(\vec{X})$ and $p^{-}(\vec{X})$ are, respectively, the definitions learned for the positive and the negative concept, obtained by renaming the positive predicate by p^{+} and its explicit negation by p^{-}. From now onwards, we will indicate with these superscripts the definitions learned separately for the positive and negative concepts.

We want $p(\vec{X})$ and $\neg p(\vec{X})$ each to act as an exception to the other. In case of contradiction, this will introduce mutual circularity, and hence undefinedness according to *WFSX*. For each literal in the intersection of p^{+} and p^{-}, there are two stable models, one containing the literal in its three-valued version, the other containing the opposite literal. According to *WFSX*, there is a third (partial) stable model where both literals are undefined, i.e., no literal $p(\vec{X})$, $\neg p(\vec{X})$, $not\ p(\vec{X})$ or $not\ \neg p(\vec{X})$ belongs to the model. This is the least partial stable model and represents the well-founded model of the theory. The resulting program contains a recursion through negation (i.e., it is non-stratified) but the top-down *SLX* procedure does not go into a loop because it comprises mechanisms for loop detection and treatment, which are implemented by *XSB* through tabling.

Note that $p^{+}(\vec{X})$ and $p^{-}(\vec{X})$ can display as well the undefined truth value, either because the original background is non-stratified or because they rely on some definition learned for another target predicate, which is of the form above and therefore non-stratified. In this case, three-valued semantics can produce literals with the value "undefined", and one or both of $p^{+}(\vec{X})$ and $p^{-}(\vec{X})$ may be undefined. If one is undefined and the other is true, then the rules above make both p and $\neg p$ undefined, since the negation by default of an undefined literal is still undefined. However, this is counter-intuitive: a defined value should prevail over an undefined one.

In order to handle this case, we suppose that a system predicate $undef(X)$ is available[2], that succeeds if and only if the literal X is undefined. So we add the following two rules to the definitions for p and $\neg p$:

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

According to these clauses, $p(\vec{X})$ is true when $p^{+}(\vec{X})$ is true and $p^{-}(\vec{X})$ is undefined, and conversely.

Contradiction on Examples Theories are tested for consistency on all the literals 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 learn clauses that cover a small amount of counter examples. This is advantageous when it

[2] The *undef* predicate can be implemented through negation NOT under CWA ($NOT\ P$ means that P is false whereas *not* means that P is false or undefined), i.e., $undef(P) \leftarrow NOT\ P, NOT(not\ P)$.

would be otherwise impossible to learn a definition for the concept, because no clause is contained in the language bias that is consistent, or when an overspecific definition would be learned, composed of very many specific clauses instead of a few general ones. In such cases, the definitions of the positive and negative concepts may cover examples of the opposite training set. These must then be considered exceptions and treated as abnormalities.

Exceptions are due to abnormalities in the opposite concept. In the latter 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 so allows.

Let us start with the case where some literals covered by a definition belong to the opposite training set. We want of course to classify these according to the classification given by the training set, by making such literals *exceptions*. To handle exceptions to classification rules, we add a negative default literal of the form $not\ abnorm_p(\vec{X})$ (resp. $not\ abnorm_{\neg p}(\vec{X})$) to the rule for $p(\vec{X})$ (resp. $\neg p(\vec{X})$), to express possible abnormalities arising from exceptions. Then, for every exception $p(\vec{t})$, an individual fact of the form $abnorm_p(\vec{t})$ (resp. $abnorm_{\neg p}(\vec{t})$) is asserted so that the rule for $p(\vec{X})$ (resp. $\neg p(\vec{X})$) does not cover the exception, while the opposite definition still covers it. In this way, exceptions will figure in the model of the theory with the correct truth value. The learned theory thus takes the form:

$$\begin{array}{rcll} p(\vec{X}) & \leftarrow & p^+(\vec{X}), not\ abnorm_p(\vec{X}), not\ \neg p(\vec{X}) & (1) \\ \neg p(\vec{X}) & \leftarrow & p^-(\vec{X}), not\ abnorm_{\neg p}(\vec{X}), not\ p(\vec{X}) & (2) \\ p(\vec{X}) & \leftarrow & p^+(\vec{X}), undef(p^-(\vec{X})) & (3) \\ \neg p(\vec{X}) & \leftarrow & p^-(\vec{X}), undef(p^+(\vec{X})) & (4) \end{array}$$

Abnormality literals have not been added to the rules for the undefined case because a literal which is an exception is also an example, and so must be covered by its respective definition; therefore it cannot be undefined.

Notice that if E^+ and E^- overlap for some example $p(\vec{t})$, then $p(\vec{t})$ is classified $false$ by the learned theory.

Individual facts of the form $abnorm_p(\vec{X})$ are then used as examples for learning a definition for $abnorm_p$ and $abnorm_{\neg p}$, as in [14, 15]. In turn, exceptions to the definitions of $abnorm_p$ and $abnorm_{\neg p}$ may be found and so on, thus leading to a hierarchy of exceptions.

These techniques have been implemented in the system LIVE: the system learns a definition for a concept and its opposite and then combines them by means of non-deterministic rules. Moreover, the system is able to identify exceptions and to learn a hierarchical definition for them.

Example 3 *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:

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

A possible learned theory is:

$$\begin{array}{rcl} flies(X) & \leftarrow & flies^+(X), not\ ab_{flies}(X), \\ & & not\ \neg flies1(X) \\ \neg flies(X) & \leftarrow & flies^-(X), not\ flies(X) \\ flies(X) & \leftarrow & flies^+(X), undef(flies^-(X)) \\ \neg flies(X) & \leftarrow & flies^-(X), undef(flies^+(X)) \\ ab_{flies}(d) & & \end{array}$$

where $flies^+(X) \leftarrow has\_wings(X)$ *and* $flies(X)^- \leftarrow has\_limbs(X)$ *Moreover, the abnormality fact* $ab_{flies}(d)$ *can be generalized to obtain*

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

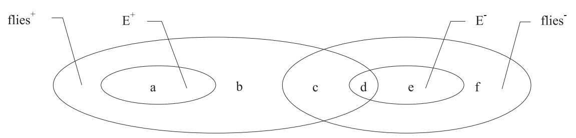

Figure 3: Coverage of definitions for opposite concepts

The example above and figure 3 show all the various cases for a literal when learning in a three-valued setting. a and e are examples that are consistently covered by the definitions. b and f are unseen literals on which there is no contradiction. c and d are literals where there is contradiction, but c is classified as undefined whereas d is considered as an exception to the positive definition and is classified as negative.

Extended logic programs can be used as well to represent n disjoint classes $p_1, \ldots, p_n$. When one has to learn n disjoint classes, the training set contains a number of facts for a number of predicates $p_1, \ldots, p_n$. Let p_i^+ be a definition learned by using, as positive examples, the literals in the training set classified as belonging to p_i and, as negative examples, all the literals for the other classes. Then the following rules ensure consistency on unseen literals and on exceptions:

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

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

4.1 Multiple Source Contradiction

In the single source case above, we showed how to deal with contradictions arising from learning conflicting rules for a predicate and its explicit negation, originating in the same knowledge source. Here we consider and handle contradictions arising from combining rules obtained from distinct knowledge sources or on distinct occasions. Let us dub it *multiple source contradiction*. This kind of situation may occur in the settings of:

- multiple, separately learning agents with distinct background knowledge, or multiple, cloned, agents with the same background knowledge;
- one agent learning separate rules from heterogenous data sources;
- one agent learning rules from uniform but separate data sets, (either because of their size, or in order to benefit from parallelism, or both);
- one agent learning separate sets of rules on different occasions;
- one agent learning separate sets of rules by employing multiple strategies or systems;
- a combination of these settings.

Generalizing the Single Source Technique The single source technique of section 4 can be easily generalized to multiple sources for learning p and $\neg p$. Let there be s sources for p and $\neg p$. We now have clauses 1-4 previously introduced, and for i from 1 to s:

$$p^+(\vec{X}) \leftarrow p_i^+(\vec{X}) \tag{5}$$

$$p^-(\vec{X}) \leftarrow p_i^-(\vec{X}) \tag{6}$$

$$abnorm_p(\vec{X}) \leftarrow abnorm_{p_i^+}(\vec{X}) \tag{7}$$

$$abnorm_{\neg p}(\vec{X}) \leftarrow abnorm_{p_j^-}(\vec{X}) \tag{8}$$

This means that whenever any two sources conflict on p for $\vec{X}$, both $p(\vec{X})$ and $\neg p(\vec{X})$ become undefined. Also, any abnormality found by one source is, *ipso facto*, an abnormality for them all. Note that some sources may provide information only about positive or negative information, thus the definition for only one of p_i^+ or p_i^- may be available.

Conflicts and Preferences However, a new situation may now arise which could not do so in the single source case: we may prefer one knowledge source over another, e.g., we may trust one source all the more because of its learning method, or because it has more recent or more trustworthy information.

To achieve this, and inspired by the program update method of [2], we generalize clause 5 and 6 above to the *combination rules*:

$$\begin{array}{lll}
p^+(\vec{X}) & \leftarrow & p_i^+(\vec{X}), not\ reject(p_i^+(\vec{X})) \\
p^-(\vec{X}) & \leftarrow & p_i^-(\vec{X}), not\ reject(p_i^-(\vec{X}))
\end{array}$$

Predicate *reject* expresses when one knowledge source, say i, is rejected by another, say j, with respect to p, through the *reject rules*[3]:

$$\begin{array}{lll}
reject(p_i^+(\vec{X})) & \leftarrow & p_j^-(\vec{X}) \\
reject(p_i^-(\vec{X})) & \leftarrow & p_j^+(\vec{X})
\end{array}$$

It may as well be the case that the positive and negative information provided by source i are rejected by two different sources k and l.

$$\begin{array}{lll}
reject(p_i^+(\vec{X})) & \leftarrow & p_k^-(\vec{X}) \\
reject(p_i^-(\vec{X})) & \leftarrow & p_l^+(\vec{X})
\end{array}$$

It can also be the case that only one or even none of these clauses is present for source i, in the case in which no source is preferred to i.

[3] If we want rejection to be as strong as what is rejected we may qualify these rules by appealing to the non undefinedness of the rejector.

But, naturally, rejection may be made to occur for a variety of reasons, and the bodies of clauses for *reject* will then observe the corresponding conditions.

As for the case of a single source, two or more knowledge sources may reject one another's conflicting conclusions. Instead of treating mutually contradictory information as undefined, as done by means of clauses 1-4, we can treat mutually contradictory information as false by means of appropriate *reject* rules, both in the single source case and in the multiple source case. Preferring *false* to *undefined* in removing a contradiction amounts to ignoring the clause instances leading to it, so that the usual CWA is adopted symmetrically with respect to positive and negative information [4].

Conflicting conclusions of two knowledge sources i and j can be made mutually *false* instead of *undefined* by means of the following instances of *reject* rules:

$$\begin{aligned} reject(p_i^+(\vec{X})) &\leftarrow p_j^-(\vec{X}) \\ reject(p_i^-(\vec{X})) &\leftarrow p_j^+(\vec{X}) \\ reject(p_j^+(\vec{X})) &\leftarrow p_i^-(\vec{X}) \\ reject(p_j^-(\vec{X})) &\leftarrow p_i^+(\vec{X}) \end{aligned}$$

If symmetry is not desired, one can remove self-contradiction by opting for only some of these clauses.

Let us now consider an example where a knowledge source is preferred over another.

Example 4 *Suppose k is the boss of i, and that they may have distinct, separately learnt, opinions about p. We may combine together their knowledge, by adding*

$$\begin{aligned} reject(p_i^+(\vec{X})) &\leftarrow p_k^-(\vec{X}) \\ reject(p_i^-(\vec{X})) &\leftarrow p_k^+(\vec{X}) \end{aligned}$$

to ensure that a conclusion arrived at by the boss wins over that of a contrary conclusion by the subordinate.

For the case of a colleague j of i, we may choose to eliminate all mutual contradictions, by means of:

$$\begin{aligned} reject(p_i^+(\vec{X})) &\leftarrow p_j^-(\vec{X}) \\ reject(p_i^-(\vec{X})) &\leftarrow p_j^+(\vec{X}) \\ reject(p_j^-(\vec{X})) &\leftarrow p_i^+(\vec{X}) \\ reject(p_j^+(\vec{X})) &\leftarrow p_i^-(\vec{X}) \end{aligned}$$

Notice that, when learning, an agent has access only to its background knowledge but, when the knowledge is combined, it may access as well the definitions of background or target predicates of other agents. In some cases it may happen that a contradiction arises exactly because, after the combination of the learned rules, an agent may use the knowledge learned by another agent as background knowledge.

Example 5 *Suppose agent i has non-contradictorily learned from examples that*

$$\begin{aligned} p_i^+(\vec{X}) &\leftarrow a(\vec{X}) \\ p_i^-(\vec{X}) &\leftarrow b(\vec{X}) \end{aligned}$$

Recall that, before knowledge sources are combined, only access to self knowledge is possible.

Further, suppose next that j has learned the rules

$$\begin{aligned} a_j(\vec{X}) &\leftarrow \neg c(\vec{X}) \\ b_j(\vec{X}) &\leftarrow \neg c(\vec{X}) \end{aligned}$$

and that the background acknowledges the fact

$$\neg c(golem)$$

When the rules from i and j are combined, i and j may access each conclusion and the background knowledge too. Now a contradiction arises in the knowledge of i regarding $p_i^+(golem)$ and $p_i^-(golem)$. If we want to resolve this contradiction by preferring false over undefined, we can use the following reject rules

$$\begin{aligned} reject(p_i^+(\vec{X})) &\leftarrow p_i^-(\vec{X}) \\ reject(p_i^-(\vec{X})) &\leftarrow p_i^+(\vec{X}) \end{aligned}$$

5 Related Work

The adoption of negation in learning has 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 [?], or decision rules, as the AQ family of systems [?], 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 undefined classification is produced, any instance is always classified as belonging to one of the classes. Instead, we classify as undefined the instances for which the learned definitions do not give an unanimous response.

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., [?, 9, 8]). 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 [9] proposed to use a three-valued logic (later on formally defined in [8]) 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. Furthermore, we are able to cope with two kinds of negation, the explicit one used to state what is false, and the default (defeasible) one used to state what can be assumed false.

The system LELP (Learning Extended Logic Programs) [14] learns extended logic programs under answer-set semantics. LELP is able to learn non-deterministic default rules with a hierarchy of exceptions. The learning problem that is presented in [14] is equivalent to the one presented in this paper when the background is a stratified extended logic program. However, when the background is a non-stratified extended logic program, 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 [?], like our *WFSX* does, and so they cannot employ any top-down proof procedure. Furthermore, answer-set semantics is not cumulative [?], i.e., if you add a lemma then the semantics can change, and thus the improvement in efficiency given by tabling cannot be obtained. 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 a literal is undefined.

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 (in 60 %-40 % range), while we always learn both concepts.

6 Conclusions

The two-valued setting that has been considered in most work on ILP and Inductive Concept Learning in general is not sufficient in many cases where we need to represent real world data. This is for example the case of an agent that has to learn the effect of the actions it can perform on the domain by performing experiments. Such an agent needs to learn a definition for allowed actions, forbidden actions and actions with an unknown outcome and therefore it needs to learn in a richer three-valued setting.

In order to adopt such a setting in ILP, the class of extended logic programs under the well-founded semantics with explicit negation (*WFSX*) is adopted as the representation language. This language allows two kinds of negation, default negation plus a second form of negation called explicit, that is used in order to represent explicitly negative information. Adopting extended logic programs in ILP prosecutes the general trend in Machine Learning of extending the representation language in order to overcome the limits of existing systems.

Contradictions may arise in the programs that are learned. We consider two cases: in the first the contradiction arises in the rules coming from a single source or agent, while in the second the contradiction arises when combining the rules coming from multiple sources or agents.

In the case of a single source, the definition for the positive and negative concept may overlap and the inconsistency is resolved in a different way for atoms in the training set and for unseen atoms: atoms in the training set are considered as exceptions, while unseen atoms are considered as unknown. The different behaviour is obtained by employing negation by default in the definitions: default abnormality literals are used in order to consider exceptions to rules, while non-deterministic rules are used in order to obtain an unknown value for unseen atoms.

The techniques used for removing contradiction from the single source case are generalized for the case of multiple sources. In this case, a predicate reject is used that expresses when a knowledge source, say i, is rejected by another, say j.

References

[1] J. J. Alferes, C. V. Damásio, and L. M. Pereira. SLX - A top-down derivation procedure for programs with explicit negation. In M. Bruynooghe, editor, *Proc. Int. Symp. on Logic Programming*. The MIT Press, 1994.

[2] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic logic programming. In *Sixth International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kauffman, 1998.

[3] J. J. Alferes and L. M. Pereira. *Reasoning with Logic Programming*, volume 1111 of *LNAI*. Springer-Verlag, 1996.

[4] J. J. Alferes, L. M. Pereira, and T. C. Przymusinski. "Classical" negation in non-monotonic reasoning and logic programming. *Journal of Automated Reasoning*, 20:107–142, 1998.

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

[6] C. V. Damásio and L. M. Pereira. Abduction on 3-valued extended logic programs. In V. W. Marek, A. Nerode, and M. Trusczynski, editors, *Logic Programming and Non-Monotonic Reasoning - Proc. of 3rd International Conference LPNMR'95*, volume 925 of *LNAI*, pages 29–42, Germany, 1997. Springer-Verlag.

[7] C. V. Damásio and L. M. Pereira. A survey on paraconsistent semantics for extended logic programs. In Gabbay and Smets, editors, *Handbook on Quantitative Reasoning: Negation and Inconsistency*, volume 2. Oxford University Press, 1998.

[8] L. De Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press, 1992.

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

[10] J. Dix, L. M. Pereira, and T. Przymusinski. Prolegomena to logic programming and non-monotonic reasoning. In J. Dix, L. M. Pereira, and T. Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming - Selected papers from NMELP'96*, number 1216 in LNAI, pages 1–36, Germany, 1997. Springer-Verlag.

[11] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *Proceedings of the 5th Int. Conf. on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[12] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of the 7th International Conference on Logic Programming ICLP90*, pages 579–597. The MIT Press, 1990.

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

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

[15] E. Lamma, P. Mello, M. Milano, and F. Riguzzi. Integrating induction and abduction in logic programming. In P. P. Wang, editor, *Proceedings of the Third Joint Conference on Information Sciences*, volume 2, pages 203–206, 1997.

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

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

[18] J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs. In J. Dix, L. M. Pereira, and T. C. Przymusinski, editors, *Collected Papers from Workshop on Logic Programming and Knowledge Representation LPKR'97*, number 1471 in LNAI. Springer-Verlag, 1998.

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

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

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

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

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

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

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

[26] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.