Empowered Negative Specialization in Inductive Logic Programming

Stefano Ferilli[1,2], Andrea Pazienza[1](✉), and Floriana Esposito[1,2]

[1] Dipartimento di Informatica, Università di Bari, Bari, Italy
{stefano.ferilli,andrea.pazienza,floriana.esposito}@uniba.it
[2] Centro Interdipartimentale per la Logica e sue Applicazioni,
Università di Bari, Bari, Italy

Abstract. In symbolic Machine Learning, the incremental setting allows to refine/revise the available model when new evidence proves it is inadequate, instead of learning a new model from scratch. In particular, *specialization* operators allow to revise the model when it covers a negative example. While specialization can be obtained by introducing negated preconditions in concept definitions, the state-of-the-art in Inductive Logic Programming provides only for specialization operators that can negate single literals. This simplification makes the operator unable to find a solution in some interesting real-world cases.

This paper proposes an empowered specialization operator for Datalog Horn clauses. It allows to negate conjunctions of pre-conditions using a representational trick based on predicate invention. The proposed implementation of the operator is used to study its behavior on toy problems purposely developed to stress it. Experimental results obtained embedding this operator in an existing learning system prove that the proposed approach is correct and viable even under quite complex conditions.

1 Introduction

Supervised Machine Learning approaches based on First-Order Logic (FOL) representations are particularly indicated in real-world tasks in which the relationships among objects play a relevant role in the definition of the concepts of interest. However, the learned theory is valid only until there is evidence to the contrary (i.e., new observations that are wrongly classified by the theory). In such a case, either a new theory is to be learned from scratch using the new batch made up of both the old and the new examples, or the existing theory must be incrementally revised to account for the new evidence as well. If positive and negative examples are provided in a mixed and unpredictable order to the learning system, two different refinement operators are needed: a *generalization* operator to refine a hypothesis that does not account for a positive example, and a *specialization* operator to refine a hypothesis that erroneously accounts for a negative example.

The focus of this paper is on supervised incremental inductive learning of logic theories from examples, and specifically on the specialization operator.

© Springer International Publishing Switzerland 2015
M. Gavanelli et al. (Eds.): AI*IA 2015, LNAI 9336, pp. 288–300, 2015.
DOI: 10.1007/978-3-319-24309-2_22

In these operators the addition of negative information may allow to learn a broader range of concepts. Following the theoretical study in [5], this paper defines an operational search space for a specialization operator that may involve multiple literals in a negation, and provides an algorithm to compute it. An implementation of the operator was embedded in the incremental learning system InTheLEx [4] and tested on purposely developed datasets.

Section 2 introduces our logic framework and the state-of-the-art specialization operator for it. Section 3 describes our new proposal and Section 4 evaluates its efficiency and effectiveness. Finally, Section 5 concludes the paper.

2 Preliminaries

Inductive Logic Programming (ILP) aims at learning logic programs from examples. In our setting, examples are represented as clauses, whose body describes an observation, and whose head specifies a relationship to be learned, referred to terms in the body. Negative examples for a relationship have a negated head. A learned program is called a *theory*, and is made up of *hypotheses*, i.e. sets of program clauses all defining the same predicate. A hypothesis *covers* an example if the body of at least one of its clauses is satisfied by the body of the example. The *search space* is the set of all clauses that can be learned, ordered by a generalization relationship.

In ILP, a standard practice to restrict the search space is to impose biases on it [10]. We are concerned with *hierarchical* (i.e., non-recursive) theories made up of linked Datalog$^\neg$ clauses. *Datalog* [1,6] is a sublanguage of Prolog in which a term can only be a variable or a constant. The missing expressiveness of function symbols can be recovered by *flattening* [12]. Datalog$^\neg$ extends pure Datalog by allowing the use of negation in the body of clauses. In the following, we will denote by $body(C)$ and $head(C)$ the set of literals in the body and the atom in the head of a Horn clause C, respectively. We adopt the Object Identity (OI) assumption: *within a clause, terms denoted by different symbols must be distinct.* This notion can be viewed as an extension of both Reiter's *unique-names* assumption [11] and Clark's Equality Theory [9] to the variables of the language. *Datalog*^{OI} is the resulting language. It has the same expressive power as Datalog [14], but causes the classical ordering relations among clauses to be modified, thus yielding a new structure of the corresponding search spaces for the refinement operators.

The ordering relation defined by the notion of θ-subsumption under OI upon Datalog clauses [3,13] is θ_{OI}-subsumption, denoted by $\leq_{OI}$. Requiring that terms are distinct, θ_{OI}-subsumption maps each literal of the subsuming clause onto a single, different literal in the subsumed one. So, equivalent clauses under $\leq_{OI}$ must have the same number of literals, and the only way to have equivalence is through variable renaming. Indeed, under OI, substitutions[1] are required to be injective.

[1] Substitutions are mappings from variables to terms [16]. Given a substitution σ by which a clause $C \leq_{OI} E$, σ^{-1} denotes the corresponding antisubstitution, i.e. the inverse function of σ, mapping some terms in E to variables in C.

The canonical inductive paradigm requires the learned theory to be complete and consistent. For hierarchical theories, the following definitions are given (where E^- and E^+ are the sets of all the negative and positive examples, resp.):

Definition 1 (Inconsistency)

- *A clause C is* inconsistent *wrt $N \in E^-$ iff $\exists \sigma$ injective substitution s.t. $body(C).\sigma \subseteq body(N) \wedge \neg head(C).\sigma = head(N)$*
- *A hypothesis H is* inconsistent *wrt N iff $\exists C \in H$: C is inconsistent wrt N.*
- *A theory T is* inconsistent *iff $\exists H \subseteq T$, $\exists N \in E^-$: H is inconsistent wrt N.*

Definition 2 (Incompleteness)

- *A* hypothesis *H is* incomplete *wrt $P \in E^+$ iff $\forall C \in H$: not($P \leq_{OI} C$).*
- *A* theory *T is* incomplete *iff $\exists H \subseteq T$, $\exists P \in E^+$: H is incomplete wrt P.*

When the theory is to be learned incrementally, it becomes relevant to define operators that allow a refinement of *too weak* or *too strong* programs [7]. *Refinement operators* are the means by which wrong hypotheses in a logic theory are changed in order to account for new examples with which they are incomplete or inconsistent. A refinement operator, applied to a clause, returns another clause that subsumes (upward refinement) or is subsumed by (downward refinement) the given clause.

Refinement operators have several applications in the automatic synthesis of logic theories. They were introduced by Shapiro [15], who used them for refining discarded hypotheses. The mathematics of refinement operators in themselves were studied by Laird [8], who described both "downward" refinement (of which Shapiro's operators were examples) and "upward" refinement. Tinkham [17] applies these ideas to allow patterns in previously-seen theories to guide the synthesis of new theories: a generalization (upward refinement) operator and a specialization (downward refinement) operator are used to form generalizations of known theories; these generalizations, together with the specialization operator, are then used to synthesize new theories.

In the following, we will assume that logic theories are made up of clauses that have only variables as terms, built starting from observations described as conjunctions of ground facts (i.e., variable-free atoms). This restriction simplifies the refinement operators for a space ordered by θ_{OI}-subsumption defined in [3, 13], and the associated definitions and properties.

Definition 3 (Refinement operators under OI) *Let C be a Datalog clause.*

- *$D \in \rho_{OI}(C)$ (*downward *refinement operator) when $body(D) = body(C) \cup \{l\}$, where l is an atom s.t. $l \notin body(C)$.*
- *$D \in \delta_{OI}(C)$ (*upward *refinement operator) when $body(D) = body(C) \setminus \{l\}$, where l is an atom s.t. $l \in body(C)$.*

The research on incremental approaches is not very wide, due to the intrinsic complexity of learning in environments where the available information about the concepts to be learned is not completely known in advance, especially in a FOL setting. Thus, [3,13] still represent the state-of-the-art for this research.

When a negative example is covered, a specialization of the theory must be performed. Starting from the current theory, the misclassified example and the previous positive examples, the specialization algorithm outputs a revised theory. In our framework, specializing means adding proper literals to a clause that is inconsistent with respect to a negative example, in order to avoid its covering that example. The possible options for choosing such a literal might be so large that an exhaustive search is infeasible.

According to Definition 3, only positive literals can be added. These literals should characterize all the past positive examples and discriminate them from the current negative one. Thus, the operator tries to specialize the clause by adding to it a literal that is present in all positive examples but not in the negative one. To satisfy the property of maximal generality, the operator must add as few atoms as possible. If no (set of) positive literal(s) is able to characterize the past positive examples and discriminate the negative example that causes inconsistency, the addition of a negative literal to the clause body might restore consistency. Such a literal should be able to discriminate the negative example from all the past positive ones. These literals cause the need to extend the representation language of clauses from Datalog to Datalog$^{\neg}$.

Consider a clause C that is inconsistent with respect to a negative example N, and the positive examples $P_1, \ldots, P_n$ that are θ_{OI}-subsumed by C. The specialization operator should restore consistency, returning a refinement C' of C which still θ_{OI}-subsumes $P_1, \ldots, P_n$, but not N.

Let us first define the *residual* of an example with respect to a clause, consisting of all the literals in the example that are not involved in the θ_{OI}-subsumption test, after having properly turned their constants into variables:

Definition 4 (Residual) *Let C be a clause, E an example, and σ_j a substitution s.t. $body(C).\sigma_j \subseteq body(E)$ under OI.*
A residual *of E wrt C under the mapping σ_j, denoted by $\Delta_j(E, C)$, is:*

$$\Delta_j(E, C) = body(E).\underline{\sigma}_j^{-1} - body(C)$$

where $\underline{\sigma}_j^{-1}$ is the *extended antisubstitution* of σ_j, defined on the whole set $consts(E)$, that associates *new* (fresh) variables to the constants in E that do not have a corresponding variable according to σ^{-1}. Each substitution by which a clause subsumes an example yields a distinct residual.

Let us define the target space for the negative literals to be added by the operator. Extending the specialization operator proposed in [3], [5] defined a new operator that selects a literal that is present in all residuals of the negative example and that is not present in any residual of any positive example:

$$l \in \mathbf{S}'_c = \neg(\overline{\mathbf{S}} - \overline{\mathbf{P}})$$

where:

$$\overline{\mathbf{S}} = \bigcap_{j=1,\ldots,m} \Delta_j(N, C)$$

$$\overline{\mathbf{P}} = \bigcup_{\substack{i=1,\ldots,n \\ j_i=1,\ldots,n_i}} \Delta_{j_i}(P_i, C)$$

and, given a set of literals $\varphi = \{l_1, \ldots, l_n\}$, $n \geq 1$: $\neg\varphi = \{\neg l_1, \ldots, \neg l_n\}$.

However, this *space of consistent negative downward refinements* does not ensure completeness with respect to the previous positive examples, as shown in the following example[2].

Example 1. ex:wrongspsoperator Consider the real world task of classifying **edible mushrooms**. A mushroom m is described by the following features: a stem s, a cap c, spores p, gills g, dots d. Two positive examples: $P_1 = m$:- $s, c, p, g.$ and $P_2 = m$:- $s, c, d.$ produce as a least general generalization the clause $C_1 = m$:- s, c. Then, the negative example $N_1 = m$:- $s, c, p, g, d.$ arrives. The residuals are $\Delta(P_1, C_1) = \{p, g\}$, $\Delta(P_2, C_1) = \{d\}$, and $\Delta(N_1, C_1) = \{p, g, d\}$. No specialization by means of positive literals can be obtained. Switching to the space of negative literals, no single literal from the negative residual, if negated, generates a clause that is complete with all previous positive examples:

$C_2' = m$:- $s, c, \neg p.$ $\quad$ $C_2'' = m$:- $s, c, \neg g.$ $\quad$ $C_2''' = m$:- $s, c, \neg d.$

where $P_1 \not\leq_{OI} C_2'$, $P_1 \not\leq_{OI} C_2''$ and $P_2 \not\leq_{OI} C_2'''$. Indeed, $\mathbf{S}_c' = (\{p, g, d\}) - (\{p, g\} \cup \{d\}) = \{p, g, d\} - \{p, g, d\} = \emptyset$. So, in this case no single (positive or negative) literal can restore completeness and consistency of the theory.

3 Extended Negative Downward Refinement

When the negation of a single atom is insufficient to restore consistency while preserving completeness, it might be the case that negating a conjunction of atoms resolves the problem. The atoms in the conjunction must be all present in the negative example, but at least one of them must not be present in any positive example. Unfortunately, these solutions are not permitted in the representation language, since only literals may appear in the body of clauses. A possible solution is to invent a new predicate defined by the conjunction, and to negate a single atom that is a suitable instance of this predicate. By a resolution step, the meaning of the resulting theory would be preserved.

Example 2. In the previous example, either $C_2' = m$:- $s, c, \neg(p, d).$ or $C_2'' = m$:- $s, c, \neg(g, d).$ would be correct refinements of C_1 wrt $\{P_1, P_2, N_1\}$. So, we might invent a new predicate n, defined as n :- $p, d.$ or n :- $g, d.$, and specialize C_1 in $C_1' = m$:- $s, c, \neg n.$. I.e., an edible mushroom must not have both spores and dots.

So, the extension comes into play when no single literal is sufficient to restore correctness of the theory. In particular, we would like to find a minimal (in terms of number of elements) such set. A formal definition of this operator is given

[2] For the sake of readability, in the following we will often switch to a propositional representation. This means that the residual is unique for each example, so the subscript in $\Delta_i(\cdot, \cdot)$ is no more necessary.

in [5]. Here, we show a possible way to define and restrict the search space, and implement a preliminary version of the operator for analysis purposes. As regards the implementation, the constraint is to determine a minimal subset of the residual, useful to specialize the clause for creating the invented predicate definition. This becomes much more relevant, but much more challenging as well, when the number of literals in the residual is very large, with very few thereof being sufficient to reach the objective. Obviously, generating and testing all possible subsets of the residual by increasingly larger cardinality, until the discriminant one is found, would ensure finding out the minimal solution. However, in the worst case this approach would require exponential time in the cardinality of the residual. We want our knowledge about the positive examples to guide our search. Specifically, we observe that no subset of literals that is present in the residual of a positive example can be used for specialization, unless other literals are included as well, otherwise that positive example would be uncovered. In the simplest case, no pair of literals that appear in the residual of a positive example can be used as a solution (i.e., as the conjunction of literals to be negated) unless another literal, not present in the residual, is added as well. So, we start considering the pairs of literals in the negative residual that are not present in any positive residual. If none of them is a solution, we must consider triplets of literals, each of which may contain at most two literals that are in the same positive residual. If again none of them is a solution, we must consider 4-tuples of literals, each of which may contain at most three literals that are in the same positive residual. And so on.

The proposed heuristic is based on the same principle but starts from the opposite perspective, proceeding top-down from the whole negative residual and progressively deriving smaller subsets thereof. Two subsets are derived from each given set of literals, based on a pair of literals in it: each subset includes one of such literals but not the other. So, the question is how to select the pairs of literals that guide the splits. Based on the previous observations, using a pair of literals belonging to different positive residuals would increase the chances of uncovering positive examples, because the remaining literals in each resulting subset tend to be concentrated in less positive residuals. Conversely, using a pair of literals that are in the same positive residual leaves in the resulting subsets more literals that belong to different positive residuals, because each of them guarantees that those two positive examples are not uncovered. So, we use the list of pairs of literals found in positive residuals as a guide for building the space of possible definitions of the invented predicate.

Algorithm 1 generates the search space for candidate definitions of an invented predicate based on this heuristic. The search space is represented as a binary tree T, made up of vertexes V and edges E, where each vertex is a candidate definition. The length of the definition (number of literals) decreases as the depth of the vertex increases, up to definitions made up of two literals (due to the strict inclusion test, the IF statement in the inner loop does not generate offspring from nodes that include just two literals). The tree is built by scanning the set R of pairs of literals that appear in at least one positive residual, and

Algorithm 1. Search space generation

```
Require: $\Delta$: negative residual; $\{R_i\}$ = residuals of positive examples
  $R \leftarrow \cup_i\{\{l', l''\} \mid l', l'' \in R_i, l' \neq l''\}$
  $T = (V, E) \leftarrow (\{\Delta\}, \emptyset)$
  $Q \leftarrow \{\Delta\}$
  while $R \neq \emptyset$ do
    $Q' \leftarrow \emptyset$
    extract $\{l', l''\}$ from $R$
    for all $L \in Q$ do
      if $\{l', l''\} \subset L$ /* if $L$ is a pair it is not split */ then
        $L_l \leftarrow L \setminus \{l'\}, L_r \leftarrow L \setminus \{l''\}$
        $Q' \leftarrow Q' \cup \{L_l, L_r\}$
        $V \leftarrow V \cup \{L_l, L_r\}$; $E \leftarrow E \cup \{(L, L_l), (L, L_r)\}$
      else
        $Q' \leftarrow Q' \cup \{L\}$
      end if
    end for
    $Q \leftarrow Q'$
  end while
Ensure: $T$
```

appending two children to each vertex in the current frontier Q (i.e., the current leaves) that includes the considered pair of literals. Each of the children removes from its parent a literal of the current pair, based on the proposed heuristic. At the beginning only the root, associated to the whole negative residual Δ, is present. Then, each loop uses an element of R to grow the offspring of the current frontier, obtaining a new frontier Q' that replaces Q in the next round.

To ensure that a minimal solution is found, one must first check if any of the vertexes made up of pairs of literals is able to restore consistency while still ensuring completeness. If no such vertex exists, the level immediately above, whose vertexes are made up of triplets of literals, is scanned, and so on until a suitable set of literals is found (in the worst case, the whole negative residual, associated to the root of the tree, is attempted), or the specialization fails. The proposed search space can be explored in depth or in breadth. In the former case, one must traverse the tree until the leaves are reached, considering only leaves associated to pairs of literals. If the whole tree has been scanned without finding a solution, these leaves are removed and the search is started again, this time focusing on the leaves associated to triplets only. And so on. In the case of a breadth search, the tree levels are explored until the 2-literal level is reached. Then, such a level is scanned looking for a complete and consistent solution. If no such solution exists, the level immediately above is tried, and so on.

In our implementation we adopted the breadth solution, but to save memory space we deleted the previous level as soon as its offspring is generated. In terms of Algorithm 1, the statement that updates V and E is dropped, so that only the vertexes in Q are available for processing. The consequence of this choice is that, if no solution can be found at the level of interest in a given round of the loop, the tree must be re-generated from the root up to the previous level in the next round of the loop. This has clearly a significant impact on runtime, but we wanted to try this solution to have an idea of the algorithm's behavior when the search space becomes too large for being stored in memory.

However, we also implemented a quick optimization to speed up the algorithm in some easy cases. Specifically, a preliminary step is introduced to check if the specialization predicate can be created by just taking a pair of literals that is present in the negative residual but not in any positive residual. Any element in this difference is a pair of literals that, being present in the negative example, if negated prevents it from being covered, and, never appearing in any positive example, does not cause any positive example to be uncovered. So, it can be used to define the invented predicate to be used for specialization without generating the whole search space up to the two-literal leaves. If the difference is empty, then no pair of literals can uncover the negative example while still covering the positive examples, so the algorithm proceeds with the normal computation for groups made up of at least three literals.

The proposed solution is as follows. After determining the set R of all unordered pairs of literals that are present in any positive residual, it is scanned and, for each pair $\{l', l''\}$ in it, the negative residual generates two shorter subsets, one containing only l' and the other containing only l''. This operation results in a binary tree, in which each branch selects a different literal for each possible solution. For each subsequent pair of literals that is considered, all leaves that contain both those literals are in turn split into two subsets. The splits go on until the tree reaches depth $|\Delta| - N$, involving solutions made up of N literals. At the beginning of the algorithm execution $N = 3$, because the solutions made up of just two literals have already been considered (and discarded) in the pre-processing step. Then, N is progressively increased by 1.

Example 3. Consider a hypothesis: $C = h$:- $a, b.$, three positive examples: $P_1 = h$:- $a, b, c, d, e.$ $P_2 = h$:- $a, b, e, f, g.$ $P_3 = h$:- $a, b, c, e, f.$, and a negative one: $N = h$:- a, b, c, d, e, f, g. The unordered pairs of literals in the residual of the negative example are computed: $\Delta(N, C) = \{c, d, e, f, g\} \rightarrow$
$S = \{\{c, d\}, \{c, e\}, \{c, f\}, \{c, g\}, \{d, e\}, \{d, f\}, \{d, g\}, \{e, f\}, \{e, g\}, \{f, g\}\}$.
For each positive residual, all unordered pairs of literals are determined:
$\Delta(P_1, C) = \{c, d, e\} \rightarrow \{\{c, d\}, \{c, e\}, \{d, e\}\}$
$\Delta(P_2, C) = \{e, f, g\} \rightarrow \{\{e, f\}, \{e, g\}, \{f, g\}\}$
$\Delta(P_3, C) = \{c, e, f\} \rightarrow \{\{c, e\}, \{c, f\}, \{e, f\}\}$
Their union is $R = \{\{c, d\}, \{c, e\}, \{d, e\}, \{e, f\}, \{e, g\}, \{c, f\}, \{f, g\}\}$, and $S \setminus R = \{\{c, g\}, \{d, f\}, \{d, g\}\}$. Any element in this difference can be used to define the invented predicate to be used for specialization.

Let us add another positive example so that no two-literal solutions exist: $P_4 = h$:- a, b, c, d, f, g. Now the set of pairs of literals from the positive residuals is:
$R = \{\{c, d\}, \{c, e\}, \{d, e\}, \{e, f\}, \{e, g\}, \{c, f\}, \{f, g\}, \{c, g\}, \{d, f\}, \{d, g\}\}$
and the difference between the two combinations is empty:
$S \setminus R = \{\{c, d\}, \{c, e\}, \{c, f\}, \{c, g\}, \{d, e\}, \{d, f\}, \{d, g\}, \{e, f\}, \{e, g\}, \{f, g\}\} \setminus$

$\{\{c, d\}, \{c, e\}, \{d, e\}, \{e, f\}, \{e, g\}, \{c, f\}, \{f, g\}, \{c, g\}, \{d, f\}, \{d, g\}\} = \emptyset$. We must use the general procedure to look for solutions that include at least 3 literals. Considering the first pair $\{c, d\}$, the two subsets are $L_l = \{c, e, f, g\}$ and $L_r = \{d, e, f, g\}$. This ensures that literals c and d are not together in any candidate final solution. The second pair is $\{c, e\}$, that is a subset of L_l (but not of L_r). So, L_l is split into $L_{ll} = \{c, f, g\}$ and $L_{lr} = \{e, f, g\}$. Since at this round we are interested in solutions involving three literals, these leaves are tried. Here the solution is indeed given by $L_{1r} = \{e, f, g\}$.

4 Evaluation

The performance of the operational procedure proposed in the previous section was evaluated while changing different parameters. First we studied the increase in runtime for finding a solution for an increasing number of literals in the residuals. Then, we studied the increase in runtime for finding a solution for both an increasing number of literals in the residuals and an increasing number of positive examples. Finally, we estimated the average runtime per literal, and then the results of the previous two phases will be compared and appropriate conclusions will be drawn. All experiments were run on a laptop endowed with a 2.5 GHz Intel i5 processor and 2GB RAM, running YAP Prolog 6.2.2 [2] on Ubuntu Linux 12.04.

The host learning system for our tests is InTheLEx [4]. Due to its behavior and to the peculiarities of its operators, InTheLEx places much burden on the generalization operator, which provides very refined clauses and clauses that are close to the theoretical least general generalization. While this makes it able to handle cases, e.g., of learning from positive examples only, this also causes the specialization operator to be fired very seldom, and often when a solution actually does not exist. Among the cases in which it is fired, in real-world domains it is often sufficient in its basic setting to find a solution. This results in a very limited chance that the proposed specialization operator based on predicate invention is used. This is however not bad. First, because specialization by negative literals is actually a last resort in learning a theory. Second, because predicate invention is a delicate activity that requires some kind of selection and cannot be performed extensively.

On 16 real-world datasets concerning document processing, each including 353 layout descriptions of first pages of scientific papers, the empowered negative specialization was necessary in so few cases that no statistical evaluation of its performance could be run. So, real-world datasets seem unsuitable to thoroughly test our methodology. For this reason, we purposely created two series of datasets that ensure that the specialization through predicate invention is fired, and that allow to suitably tune both the number of literals in the residuals and the number of positive examples as needed to test the approach in increasingly stressing conditions for the operator.

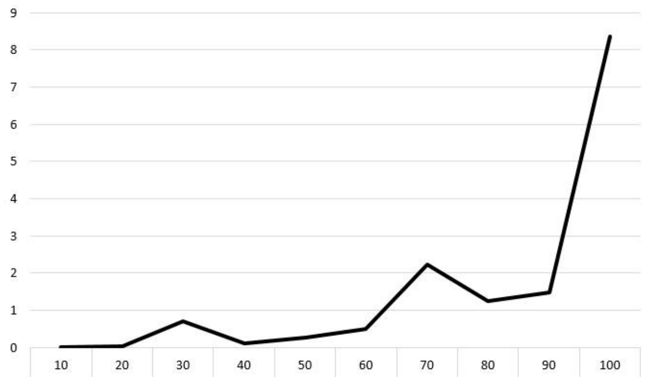

Fig. 1. Runtime (in minutes, y-axis) by number of literals in the residuals (x-axis).

4.1 Efficiency vs Residual Length

The first test involved 50 datasets, 5 for each of 10 different settings. Each dataset included 7 positive and 1 negative examples. The i-th setting ($i = 1, \ldots, 10$) produces datasets that involve $10i$ literals in the residual of the negative example: i.e., 10 literals for the first setting, 20 literals for the second setting, and so on. This is obtained as follows: a clause is created, and its body instantiated in all (positive and negative) examples. Then, $10i$ additional literals are appended to the negative example (to form its residual), and each positive residual is obtained by randomly selecting 80% of the literals from the negative residual, ensuring that each positive residual is different from the others. This should result in a larger search space for the predicate invention-based specialization algorithm. Each of the 5 datasets for each setting involves different combinations of literals in the positive examples, to provide more varied testbeds.

InTheLEx was run on each dataset, fed with all the positive examples and the clause generalizing them. The averages for the various settings are summarized in Figure 1 which shows the trend for larger and larger negative residuals that is clearly increasing, albeit not monotonically. Another interesting observation concerns the change in slope of the curve that appears around the 90-literals case. Up to 60 literals, a solution is found in less than one minute on average. Runtime is still acceptable (never above 3 minutes) from 70 to 90 literals. Instead, a sudden increase happens for 100 literals (even if the average is still below 9 minutes). A possible explanation is that a larger number of literals may cause a significant extension of the search space, resulting in much higher runtime if a solution made up of a few literals is not found. Indeed, an observation of the learned theories revealed that the number of literals that make up the invented predicate is less than half the number of literals in the residual on average.

Table 1. Runtimes for increasingly larger negative residuals

Setting	1	2	3	4	5	6	7	8	9	10
Positive Examples	7	9	11	13	15	17	19	21	23	25

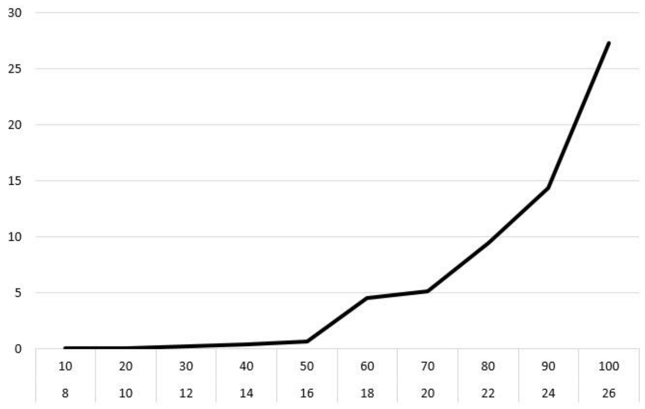

Fig. 2. Runtime (in minutes, y-axis) by number of literals in the residuals and number of examples for each setting (x-axis).

4.2 Efficiency vs Residual Length and Number of Positive Examples

Another 50 artificial datasets were created for the second kind of evaluation, using the same strategy as before except that also the number of positive examples was linearly increased in each setting, as reported in Table 1. Again, each positive example is obtained as an instance of the initial clause by adding a residual made up of 80% randomly selected literals from the negative residual, ensuring that no two combinations are the same.

The corresponding results of runtime for InTheLEx are graphically summarized in Figure 2. Up to 50 literals (and 15 positive examples) runtime is still below one minute. This is pretty good, because the number of positive examples at 50 literals in the residual is more than doubled compared to the previous experiment. For the 60 and 70 settings runtime increases around 5 minutes, and stays below 15 minutes up to the 90 literals setting, where the number of positive examples is three times as much as the previous experiment. For the 100 literals & 25 positive examples setting, runtime is around 27 minutes.

Comparing the two graphics in Figures 1 and 2, we note that in both cases runtime is below one minute up to the 50 literals setting. This means that neither the increase in the number of examples, nor that in the number of literals, significantly affects performance. The two curves part away starting from the 60 literals setting. Both go up, but the latter much more than the former. Summing up, both parameters affect runtime only after a given complexity is reached, and the number of examples has a heavier impact than the number of literals. One possible explanation is that, after that complexity level is reached, the search

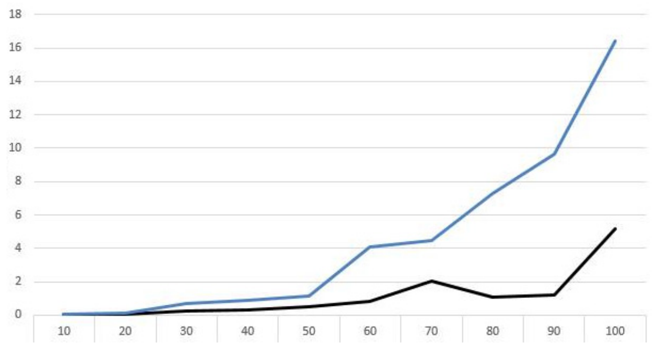

Fig. 3. Runtime (y-axis) per literal (x-axis) trend comparison.

space becomes inherently more complex, not allowing easy/small solution by its very structure. Or, at least, this is what seems to happen in the randomly generated toy problems. It will be interesting to check whether in real-world domains the non-randomness in descriptions avoids such a behavior.

The average runtime per literal for the various settings in the two experiments is reported in Figure 3. As expected, due to more literals causing more combinations to be potentially tested, the runtime is not constant. However, again, the number of examples seems to be the cause of the largest increase in time. While the number of possible literal combinations is determined by the negative residual, the branching factor in the tree is determined by the number of (positive) examples. Thus, once again it seems that the hard part is somehow related to the dataset structure: indeed, this behavior can be explained by the fact that the more positive examples, the more branches must be generated before finding the solution. Manual checking of sample cases in each setting has confirmed that the solution found by the proposed procedure is always minimal.

5 Conclusions and Future Work

Incremental supervised Machine Learning approaches using First-Order Logic representations are mandatory when tackling complex real-world tasks, in which relationships among objects play a fundamental role. A noteworthy framework for these approaches is based on the space of Datalog Horn clauses under the Object Identity assumption, which ensures the existence of refinement operators fulfilling desirable requirements, unless they have some limitations that this paper aims at overcoming. After recalling the most important elements of the framework and of the current downward operator, this paper proposed and implemented an algorithm for multi-literal negation-based specialization. The efficiency of the operator, integrated in the InTheLEx learning system, was tested using purposely devised experiments.

Future work includes a study of the possible connections of the extended operator with related fields of the logic-based learning, such as deduction, abstraction and predicate invention. Experiments aimed at assessing the efficiency and effectiveness of the operator in real-world domains are also planned.

Acknowledgments. This work was partially funded by the Italian PON 2007-2013 project PON02_00563_3489339 'Puglia@Service'.

References

1. Ceri, S., Gottlöb, G., Tanca, L.: Logic Programming and Databases. Springer-Verlag, Heidelberg (1990)
2. Costa, V.S., Rocha, R., Damas, L.: The YAP Prolog system. Theory and Practice of Logic Programming **12**(1–2), 5–34 (2012)
3. Esposito, F., Laterza, A., Malerba, D., Semeraro, G.: Locally finite, proper and complete operators for refining datalog programs. In: Michalewicz, M., Raś, Z.W. (eds.) ISMIS 1996. LNCS, vol. 1079, pp. 468–478. Springer, Heidelberg (1996)
4. Esposito, F., Semeraro, G., Fanizzi, N., Ferilli, S.: Multistrategy Theory Revision: Induction and Abduction in INTHELEX. Machine Learning Journal **38**(1/2), 133–156 (2000)
5. Ferilli, S.: Toward an improved downward refinement operator for inductive logic programming. In: Atti del 11th Italian Convention on Computational Logic (CILC-2014), vol. 1195, pp. 99–113. Central Europe (CEUR) Workshop Proceedings (2014)
6. Kanellakis, P.C.: Elements of relational database theory. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Formal Models and Semantics, vol. B, pp. 1073–1156. Elsevier Science Publishers (1990)
7. Komorowski, J., Trcek, S.: Towards refinement of definite logic programs. In: Raś, Z.W., Zemankova, M. (eds.) ISMIS 1994. LNCS, vol. 869, pp. 315–325. Springer, Heidelberg (1994)
8. Laird, P.D.: Inductive inference by refinement. In: Proc. of AAAI-1986, Philadelphia, PA, pp. 472–476 (1986)
9. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer-Verlag, Berlin (1987)
10. Nédellec, C., Rouveirol, C., Adé, H., Bergadano, F., Tausend, B.: Declarative bias in ILP. In: de Raedt, L. (ed.) Advances in Inductive Logic Programming, pp. 82–103. IOS Press, Amsterdam, NL (1996)
11. Reiter, R.: Equality and domain closure in first order databases. Journal of the ACM **27**, 235–249 (1980)
12. Rouveirol, C.: Extensions of inversion of resolution applied to theory completion. In: Inductive Logic Programming, pp. 64–90. Academic Press (1992)
13. Semeraro, G., Esposito, F., Malerba, D.: Ideal refinement of datalog programs. In: Proietti, M. (ed.) LOPSTR 1995. LNCS, vol. 1048, pp. 120–136. Springer, Heidelberg (1996)
14. Semeraro, G., Esposito, F., Malerba, D., Fanizzi, N., Ferilli, S.: A logic framework for the incremental inductive synthesis of datalog theories. In: Fuchs, N.E. (ed.) LOPSTR 1997. LNCS, vol. 1463, pp. 300–321. Springer, Heidelberg (1998)
15. Shapiro, E.Y.: Inductive inference of theories from facts. Technical Report Research Report 192, Yale University (1981)
16. Siekmann, J.H.: An introduction to unification theory. In: Banerji, R.B. (ed.) Formal Techniques in Artificial Intelligence - A Sourcebook, pp. 460–464. Elsevier Science Publisher (1990)
17. Tinkham, N.L.: Schema induction for logic program synthesis. Artif. Intell. **98**(1–2), 1–47 (1998)