Automatic Induction of First-Order Logic Descriptors Type Domains from Observations

Stefano Ferilli, Floriana Esposito, Teresa M.A. Basile, and Nicola Di Mauro

Department of Computer Science, University of Bari, Italy
{ferilli,esposito,basile,nicodimauro}@di.uniba.it

Abstract. Successful application of Machine Learning to certain real-world situations sometimes requires to take into account relations among objects. Inductive Logic Programming, being based on First-Order Logic as a representation language, provides a suitable learning framework to be adopted in these cases. However, the intrinsic complexity of this framework, added to the complexity of the specific application context, often requires pure induction to be supported by various kinds of meta-information on the domain itself and/or on its representation in order to prune the search space of all possible definitions. Indeed, avoiding the exploration of paths that do not lead to any correct solution can greatly reduce computational times, and hence becomes a critical issue for the performance of the whole learning process. In the current practice, providing such information is often in charge of the human expert. It is also a difficult and error-prone activity, in which mistakes are highly probable because of a number of factors. This makes it desirable to develop procedures that can automatically generate such information starting from the same observations that are input to the learning process.
This paper focuses on a specific kind of meta-information: the *types* used in the description language and their related *domains*. Indeed, many learning systems known in the literature are able to exploit (and sometimes require) such a kind of knowledge to improve their performance. An algorithm is proposed to automatically identify types from observations, and detailed examples of its behaviour are given. An evaluation of its performance in domains with different characteristics is reported, and its robustness with respect to incomplete observations is studied.

Keywords

Description Languages, Knowledge Acquisition

1 Introduction

Learning in particular contexts, characterized by a high degree of complexity, often requires pure induction to be supported by a variety of techniques that can cope with different aspects of the learning task. A way to overcome such a limitation is the use of other kinds of inferences in support of induction, according to an integrated framework that tries to emulate the human way of

reasoning [11]. For instance, some learning systems can take great advantage from meta-information on the domain itself and/or on its representation, in order to prune the search space and focus on the parts of it in which a solution is more likely to be found.

In the current practice, it is in charge of the human expert to specify all the 'added-value' information needed by such techniques for being applicable. It goes without saying that quality, correctness and completeness in the formalization of such meta-information is a critical issue, that can determine the very feasibility of the learning process. Providing it is a very difficult task, also, that requires a deep knowledge of the application domain, and is in any case an error-prone activity, since omissions and errors may take place for a number of reasons. For instance, the domain and/or the language used to represent it might be unknown to the experimenter, because he is just in charge of properly setting and running the learning system on a dataset provided by third parties and/or generated by other people (or even by automatic systems), as in the case of classical machine learning datasets available on the Internet. In any case, it is often not easy for non-experts to single out and formally express such meta-knowledge in the form of parameters or other kinds of representations needed by the automatic systems, just because they are not familiar with the representation language and the related technical issues. Other possible causes include the great number of such parameters to be specified, and the fact that they are sometimes hidden from the normal focus-of-attention when reasoning on the problem.

These considerations would make it highly desirable to develop procedures that can automatically generate such information starting from the same observations that are input to the learning process. Hence, a strong motivation for the research presented in this paper, aimed at proposing algorithms to automatically infer the meta-information required by the techniques referred to above, and at assessing the validity and performance of the corresponding procedures. The challenge in this attempt is that it does not try to learn something *about* the given instances, but instead aims at gathering information on the domain and/or its description *from* the given instances. This means that we are no more concerned with the description of concepts by proper juxtaposition of literals, but rather with the meaning underlying the language used. Thus, the problem can be seen as a higher-order learning, that deals with semantics rather than with syntax.

The next section presents a technique to automatically infer the description language from the very same data used by the learning procedure, and motivates the interest of researchers for having available this kind of meta-information. Then, Sections 3 and 4 extensively test the proposed approach, even in the case of incomplete input information. Last section concludes the paper and outlines future work.

2 Inducing Descriptors Type Domains

One of the most interesting issues, when dealing with observations in an unknown language, is the identification of what properties are used to describe the available knowledge, and what are the possible values for each property. In other words, there is a need to know what are the *types* used in the description language and their related *domains*. These issues are clearly strongly related to work on ontologies, where we are concerned with the representation of concepts along with their interrelationships and properties/domains. However, most of the current research in that field is devoted to devise methodologies and techniques for representing and exploiting ontological information (that in the current practice is tipically entered by human experts), while the issue of automatically learning and refining such information has not yet been takled thoroughly (also due to the inherent complexity of the task).

On the other hand, various learning systems in the literature can exploit meta-information of this kind, if available, to prune the search space and obtain this way more efficiency. Just to cite the major ones, MIS [14], FOIL [3], Progol [13] and Mobal [12] allow the experimenter to specify the type of arguments in the predicates of the description language. A straightforward exploitation of such kind of information is to carry out a pruning of the search space, that eliminates hypotheses that contrast with the descriptor meaning. This would turn out to be particularly useful in the case of incomplete descriptions, where the presence of particular properties can be only abductively guessed and the availability of integrity constraints of this kind may be a valuable help for driving such guesses. For instance, knowing that *male* and *female* are two different values of one domain, would allow a learning system to avoid generating and testing clauses that refer both values to the same individual. It is clear that, along with the complexity of the representation language and the number of properties and values used, significant amounts of impossible hypotheses could be recognized in this way and discarded in advance.

Some attempts to automatically infer such information have already been carried out: in [10], for instance, the Authors exploit the occurrence, in the available observations, of the same specific value in different positions to identify predicate arguments to be filled with information of the same type. In this case, the domain of a type can be inferred by simply collecting all the constants that appear in the predicate positions associated to that type. However, some issues about type inference still remain unsolved. Indeed, theories learned by many systems are constant-free, and allow only variables as terms. In those cases, the information expressed by constants can be recovered by a process of *reification*, that (loosely speaking) transforms *common nouns* into predicates, and uses constants as just *proper nouns* for specific objects. For instance, the property *color(bicycle,red)* is translated into *color(b,r),bicycle(b),red(r)*, this way being able to express the fact that an object is red as *color(X,Y),red(Y)*.

This is usual when using First Order Predicate Logic as a representation language: n-ary ($n > 1$) predicates are used to express relationships among different objects. Specifically, binary predicates, in addition to expressing a re-

lationship between two objects, can be exploited to associate objects with the corresponding values for their relevant (atomic) properties. In order to avoid such ambiguity, a process of *reification* can be carried out on symbolic properties, so that each possible value for them becomes itself a property, and hence can be represented by a unary predicate. This obviously breaks the association of specific values and types to particular predicate arguments, which makes it more difficult to collect them according to the properties they refer to. In such a situation, discovering the type domains for the properties in the language can be cast as the search for groups of unary predicates that semantically refer to the same attribute. Knowing this kind of types could be of great help for limiting the search space in some systems that are not strongly observation-driven: systems like Claudien [4], Primus [7] and Tertius [8], for instance, could avoid generating clauses in which more than two values belonging to the same type are associated to one object, thus dramatically limiting the combinatorial explosion in clause generation.

Example 1. Given the following examples and descriptions:
not(target(a)) :- part_of(a,a′), high(a′), large(a′), white(a′),
part_of(a,a″), low(a″), small(a″), white(a″), to_right(a′,a″).
target(b) :- part_of(b,b′), high(b′), small(b′), blue(b′),
part_of(b,b″), low(b″), large(b″), red(b″), on_top(b′,b″).
target(c) :- part_of(c,c′), high(c′), small(c′), yellow(c′),
part_of(c,c″), high(c″), large(c″), red(c″), to_right(c′,c″).
the set of unary predicates in the description language is:

{*high*, *large*, *white*, *low*, *small*, *blue*, *red*, *yellow*}

We would like the system to understand that the values they represent define three domains referred to different types, according to the following groups:

- {*white*, *blue*, *red*, *yellow*} (that, being able to catch their semantics, could be recognized as belonging to property *color*),
- {*small*, *large*} (that are related to property *size*), and
- {*high*, *low*} (related to property *height*).

In the rest of this presentation, the following (clearly non restricting) assumptions will be made:

- all non-numeric (symbolic or discretized) properties are reified, so that all of their possible values are expressed by means of unary predicates[1];
- there is no overloading on unary predicates, i.e. no unary predicate expresses a value that belongs to many types (this is a typical feature of strongly typed programming languages, e.g. for enumerative types);
- there are no 'boolean' properties, i.e. properties that are expressed by just the presence or absence of a corresponding predicate (e.g., *found*, would require an opposite predicate *not_found*);

[1] This is the actual problematic situation, since if the property values were associated to objects as arguments of (e.g., binary) predicates, the predicates themselves would be sufficient to separate and semantically identify the type domains.

– all properties are applicable to any object that occurs in the descriptions (in case this does not hold, it is sufficient to include an additional *not_applicable* value to each property that makes sense for some objects only)[2].

The first consideration one can do is that different values for the same attribute are mutually exclusive, since one given object cannot have two of them at the same time (e.g., an object can be black or blue, but not both). Hence, the first problem to be solved is finding all couples of predicates that are mutually exclusive, i.e. never co-occur referred to the same object in the available knowledge of the world (let us call them *constraints*[3]). Such information can be obtained as follows: after collecting the set of all unary predicates used in the available example descriptions, all the possible pairs of such predicates (without regard to the order) are generated, associated with the same variable as argument and then tested for occurrence in the observations themselves.

Example 2. The set of unary predicates identified in the previous example yields 56 possible pairs to be tested. Among these, some occur in the available observations, and hence the corresponding predicates cannot belong to the same type domain. For instance:

$\langle high(X),\ large(X)\rangle$, $\langle large(X),\ white(X)\rangle$, $\langle high(X),\ white(X)\rangle$

are verified by object a'. Others never occur, some including values that actually belong to the same semantic domain:

$\langle high(X),\ low(X)\rangle$, $\langle large(X),\ small(X)\rangle$, $\langle red(X),\ white(X)\rangle$, ...

and others including values that belong to different ones:

$\langle small(X),\ red(X)\rangle$, $\langle large(X),\ yellow(X)\rangle$, $\langle low(X),\ blue(X)\rangle$, ...

It goes without saying that finding mutually exclusive couples is not sufficient: More precisely, *any* value in a given domain cannot co-occur in one object with *any* other value in the same domain. Thus, the problem becomes identifying groups of unary predicates whose elements are *couplewise* mutually exclusive. In particular, since for any set of predicates fulfilling such property it holds that all of its subsets fulfill the same property as well, we are interested in maximal sets only, i.e. we discard groups that are subsets of other groups. This can be obtained by mapping the problem onto a corresponding one in the graph context. Specifically, we build an undirected graph G_e whose nodes are unary predicates in the description language, and where an edge connects two nodes if and only if they are mutually exclusive. In such a setting, the maximal sets we are looking for correspond to all the *maximal* cliques (i.e., cliques that cannot be further extended) in G_e.

[2] For instance, when describing books, properties *weight* and *price* would make sense only for the book as a whole, and not for its layout/content components such as title, foreword, etc. In such a case, the domains of the above properties (e.g., {*weight_light*, *weight_medium*, *weight_heavy*} and {*cheap*, *expensive*}) could be extended by two additional values *weight_not_applicable* and *price_not_applicable*, respectively.

[3] This notion of *constraint* can be extended to the case of n-tuples of predicates that never occur together referred to the same object.

Example 3. Applying the above technique to the set of unary predicates and mutually exclusive pairs obtained in previous examples, the following cliques (i.e., groups of pairwise mutually exclusive values) are found:

- {*blue*, *large*, *yellow*}
- {*blue*, *low*, *yellow*}
- {*blue*, *red*, *white*, *yellow*}
- {*high*, *low*}
- {*large*, *small*}
- {*red*, *small*}

The groups found this way are still far from being the desired solutions. Indeed, there can be groups of predicates with couplewise mutually exclusive elements even if they do not refer to a same attribute. For instance, it is generally true that a line is never too tall, hence in a paper document domain we might find the group {*line*, *high*, *very_high*, *highest*} in which it is obvious that value *line* belongs to the domain of type *shape*, while the other three values refer to the type *height*. Nevertheless, we expect that two correct (i.e. distinct, or, more precisely, *disjoint*) groups exist, one containing all (and only those) values belonging to property *shape*, and the other containing all (and only those) values belonging to property *height*. Here, the clue is that, in the end, the desired solution will include only groups that have no element in common. Hence, since the above group would have elements in common with properties *height* and *shape*, it should be discarded. Again, this problem can be solved in the graph context by building an undirected graph G_d in which nodes are groups identified in the previous step as cliques of graph G_e, and an edge connects two nodes if and only if they are disjoint sets. Now, the solution will be made up by only couplewise disjoint subsets, and specifically by maximal groups of disjoint subsets, each of which corresponds to a maximal clique in G_d.

Example 4. Continuing with the previous examples results, the clique technique returns the following sets of possible mutually disjoint groups of predicates:

- { {*blue*, *large*, *yellow*}, {*high*, *low*}, {*red*, *small*} } (including 7 values)
- { {*blue*, *low*, *yellow*}, {*large*, *small*} } (including 5 values)
- { {*blue*, *low*, *yellow*}, {*red*, *small*} } (including 5 values)
- { {*blue*, *red*, *white*, *yellow*}, {*high*, *low*}, {*large*, *small*} } (including 8 values)

As in the example above, the clique in G_d will probably not be unique, in which case one must have a clue for choosing the right one. The intuition, in this case, is that any 'wrong' clique, in order to fulfill the mutual disjunction requirement, will have overall a number of values that is less than that of the correct solution, since the correct solution should be the only one containing all the possible values for each property (represented by a group), and hence the union of predicates in all of its components should be equal to the whole set of values for all possible attributes. In other words, the solution is actually a *partition* of the set of unary predicates. This holds because the description

Algorithm 1 Identification of type domains

Require: L: Description language
1: $U := \{p \in L \mid p \text{ unary}\}$
2: $E := \{(p,q) \in U \times U \mid \nexists X : p(X) \wedge q(X)\}$
3: $G_e := (U, E)$
4: $S := \{C \subseteq U \mid C \text{ clique in } G_e\}$
5: $F := \{(p,q) \in S \times S \mid p \cap q = \emptyset\}$
6: $G_d := (S, F)$
7: $T := \{C \subseteq S \mid C \text{ clique in } G_d\}$
Ensure: $argmax_{t \in T}(|\bigcup_{t_i \in t} t_i|)$: type domains

language is assumed not to contain 'boolean' properties; if it does, the union would not be a partition, but should in any case contain more unary predicates than any other candidate partition.

Example 5. Among the sets of disjoint groups identified in the previous step, the only one containing all 8 unary predicates in the description language is { {*blue*, *red*, *white*, *yellow*}, {*high*, *low*}, {*large*, *small*} }, that also corresponds to the solution, as expected.

The whole strategy is summarized in Algorithm 1. Given a set of examples along with their descriptions, all unary predicates (corresponding to property values) in the description language are collected, and a graph is built having such predicates as nodes and an edge between two nodes if the corresponding predicates are never referred to the same object in the available observations. Then, the cliques in such a graph are identified, and used as nodes for building another graph, whose edges connect two nodes if they share no predicate. Lastly, for each clique in this latter graph the union of the sets of predicates in the nodes is computed, and the one with the greatest cardinality is chosen as the solution. $argmax_{t \in T}(|\bigcup_{t_i \in t} t_i|)$ corresponds to the clique t in G_d that maximizes the cardinality of (i.e., the number of predicates in) the set union of its composing nodes t_i.

It is presented in a simple and linear fashion, since it includes two clique computations (for which algorithms are known in the literature and often implemented as standard libraries in programming languages compilers) in steps 4 and 7, and other trivial operations: steps 3 and 6 consist of a simple name assignment to graphs, step 1 requires to scan the descriptors used in the given observations just once in order to collect the corresponding unary predicates; steps 2 and 5 collect all couples of graph nodes without regard to the order (which can be done in $\mathcal{O}(m^2)$ steps for m nodes) and test a condition on each of them. Informally speaking, the problem resembles a puzzle, in which more pieces than necessary are provided, such that the additional pieces can partly fit the others, but when added will always prevent reaching a complete solution. A consideration is worth. The feasibility of reaching the target solution requires that the number of values for the domains to be identified and the amount of available knowledge about observations to be strictly proportional. Indeed, the

more the values, the more the possible interrelations that can take place between them. If the available observations are not sufficiently significant, i.e. too many existing interrelations are not recognizable in them, then knowledge about the actual biases in the given domain would be too loose for the algorithm to properly separate semantically different values.

Example 6. Given the following problem setting:
not(target(a)) :- part_of(a,a′), high(a′), large(a′), blue(a′),
part_of(a,a″), low(a″), small(a″), white(a″), to_right(a′,a″).
target(b) :- part_of(b,b′), high(b′), small(b′), black(b′),
part_of(b,b″), low(b″), large(b″), red(b″), on_top(b′,b″).
target(c) :- part_of(c,c′), high(c′), small(c′), yellow(c′),
part_of(c,c″), high(c″), large(c″), red(c″), to_right(c′,c″).
two different partitions, both involving all 9 unary predicates {*high*, *large*, *blue*, *low*, *small*, *white*, *black*, *red*, *yellow*} could be found, and specifically:

- { {*black*, *blue*, *red*, *white*, *yellow*}, {*high*, *low*}, {*large*, *small*} }
- { {*black*, *large*, *white*, *yellow*}, {*blue*, *red*, *small*}, {*high*, *low*} }

which provides no clue for understanding which is the correct one.

As to the practical implementation of the proposed algorithm, some considerations can be made that are useful to restrict the search space of candidate disjoint predicate groups. First, note that when all unary predicates in the description language represent values of only one type, they are all mutually exclusive, thus the graph G_e is completely connected and yields just one candidate group, which coincides with the only element of the singleton partition to be found (thus, there is no need for computing G_d). Let us now face the case in which at least two types are present. Here, a way to restrict the range of possibilities to be checked is inspired to a well-known mathematical trick, that runs more or less as follows. "*A man wakes up early in the morning, and has to dress up to go work; since he cannot turn on the light not to wake up his wife, he has to pick his socks from the drawer in the dark; knowing the drawer contains n_i pairs of socks for each color $i = 1, ..., m$, how many socks should he draw at random to be sure that at least two of them are of the same color?*" The answer is, clearly, $m + 1$ (the number of colors plus one), so that the set of socks will contain at least a double. In our case, a procedure (sketched in Algorithm 2) is implemented that progressively returns the pairs $(n_k, m_k)_{k=2,3,...}$, where n_k is the number of unary predicate k-tuples that are new *constraints* (i.e., constraints that are not a superset of a previous constraint found at a step $j < k$), and m_k is the number of those that are not constraints, according to the available observations, until a $(n_{\overline{k}}, m_{\overline{k}})$ is found such that $(n_{\overline{k}+1}, m_{\overline{k}+1}) = (0, 0)$.

Then, only the cliques of size (greater than or) equal to $\overline{k}$ must be taken into account. Indeed, if the number of types is $\overline{k}$, any group of unary literals of size $\overline{k} + 1$ will contain at least 2 values taken from one type domain. Thus, these two literals have surely appeared previously as binary constraints, and hence no (new) constraint nor non-constraint will be present of size $\overline{k} + 1$, i.e.

Algorithm 2 Identification of the maximum number of domains

Require: U: Unary predicates in the description language, O: Available observations
  $NonConstraints_1 \leftarrow \{\{p\}|p \in U\}$
  $n_1 \leftarrow 0$; $m_1 \leftarrow |NonConstraints_1|$
  $k \leftarrow 1$
  while $\neg(n_k = 0 \wedge m_k = 0)$ **do**
    $k \leftarrow k + 1$
    $NonConstraints_k \leftarrow \emptyset$
    $Constraints_k \leftarrow \emptyset$
    for all $N \in NonConstraints_{k-1}$ **do**
      for all $p \in U$, $p \notin N$ **do**
        if $N' = N \cup \{p\}$ is verified by some object in O **then**
          $NonConstraints_k \leftarrow NonConstraints_k \cup \{N'\}$
        else
          if $\nexists C \in Constraints_j, j < k \ni' C \subset N'$ **then**
            $Constraints_k \leftarrow Constraints_k \cup \{N'\}$
          end if
        end if
      end for
    end for
    $n_k \leftarrow |Constraints_k|$; $m_k \leftarrow |NonConstraints_k|$
  end while
Ensure: k: number of domains in O

$(n_{\overline{k}+1}, m_{\overline{k}+1}) = (0, 0)$. Conversely, for $k \leq \overline{k}$ there will surely be a k-tuple of unary predicates, in which each unary predicate is taken from a different domain. Such a k-tuple is either a (superset of a previous) constraint, or it is verified by the available observations, in which case it is a non-constraint (and thus $m_k > 0$).

Example 7. Given the following examples and descriptions:

not(target(a)) :- part_of(a,a′), large(a′), part_of(a,a″), small(a″).
target(b) :- part_of(b,b′), small(b′), blue(b′), part_of(b,b″), large(b″), red(b″).
target(c) :- part_of(c,c′), small(c′), part_of(c,c″), large(c″), red(c″).

the set of unary predicates in the description language is:

$$U = \{large, small, blue, red\}$$

We would like the procedure to infer the number $\overline{k}$ of types. We first set:

$NonConstraints_1 = \{\{large\}, \{small\}, \{blue\}, \{red\}\}$,
$n_1 = 0$ and $m_1 = |NonConstraints_1| = 4$.

Since $m_1 \neq 0$, each singleton in $NonConstraints_1$ must be extended in all possible ways by means of a new unary predicate in the description language, thus the procedure generates the following couples:

$\{large, small\}, \{large, blue\}, \{large, red\}$,
$\{small, blue\}, \{small, red\}, \{blue, red\}$.

By testing each of such couples against the available observations, the procedure generates the following sets:

$NonConstrainst_2 = \{\{small, blue\}, \{large, red\}\}$

since in the observations object b' is both small and blue, and there are two objects (b'' and c'') that are large and red at the same time.

$Constraints_2 = \{\{large, blue\}, \{small, red\}, \{large, small\}, \{blue, red\}\}$.
since no observed object is at the same time large and blue, nor small and red, nor large and small, nor blue and red.
Now, since $n_2 = |Constrainst_2| = 4 \neq 0$ and $m_2 = |NonConstrainst_2| = 2 \neq 0$ the procedure tries to extend each couple in $NonConstrainst_2$ by adding a new unary predicate. But each triple that can be obtained in this way is a superset of an element of $Constraints_2$, thus we will have $NonConstraints_3 = Constraints_3 = \emptyset$ and hence $n_3 = m_3 = 0 \Rightarrow \overline{k} = 2$.

Not only this avoids the computational overhead of handling those with lesser size, since they will not be part of the solution, but can be used also to recognize that the available data are not sufficient to carry out the desired task. Indeed, if no cliques (i.e., partitions) of size (i.e., cardinality) at least $\overline{k}$ exist in the graph, the procedure can warn the user that information in the observations is probably too loose to allow the reconstruction of the types in the representation language used.

3 Experimental Results

The proposed method was implemented in SICStus Prolog, and tested on various domains, suitably chosen in order to cover all the possible cases of available observations and target types to be recognized. Here, we report the results obtained on one sample dataset for each case.

The Scientific Papers dataset [6] is based on a representation language made up of predicates with various arities, of which unary predicates represent values belonging to many different domains (*general* case). It includes 112 scientific papers, belonging to 4 different classes (Springer-Verlag Lecture Notes, Proceedings of the International Conference on Machine Learning, IEEE Transactions, and none of the above) whose layout structure was described in terms of its composing layout blocks features (height, width, horizontal position, vertical position, content type) and relative position (horizontal adjacency, vertical adjacency, horizontal alignment, vertical alignment). The procedure reached $\overline{k} = 5$, and found the following (correct) types:

1. *Width*: {large, medium, medium_large, medium_small, small, very_large, very_small}
2. *Content*: {graphic, hor_line, image, mixed, text,ver_line}
3. *Vertical position*: {lower, middle, upper}
4. *Horizontal position*: {center, left, right}
5. *Height*: {large, medium, medium_large, medium_small, small, smallest, very_large, very_small, very_very_large, very_very_small}

The Family Relationships dataset [2] refers to a description language made up of predicates with various arities, of which unary predicates all belong to the

same type. It describes a hypothetical family in terms of each person's sex and of the basic relations among persons (parent and married), whose members' pairs are tagged according to the derived relations (father, mother, son, daughter, uncle, aunt, etc.). In this case, all the unary predicates fell in one group (thus there was no need for building G_d), that was also the only type (successfully retrieved by the algorithm):

1. *Sex*: {female, male}

The Multiplexer dataset [5] describes 6-bit configurations, with the aim of inducing the definition of a multiplexer such that, among the last four bit positions, the position denoted by the first two bits must be 1. All 64 possible bit configurations are included, which should make significantly easier the type induction task, as confirmed by the algorithm output:

1. *Sixth bit*: {bit6at0, bit6at1}
2. *Fifth bit*: {bit5at0, bit5at1}
3. *Fourth bit*: {bit4at0, bit4at1}
4. *Third bit*: {bit3at0, bit3at1}
5. *Second bit*: {bit2at0, bit2at1}
6. *First bit*: {bit1at0, bit1at1}

The Tic Tac Toe dataset [1] description language is made up of unary predicates only (representing values of different types). It contains all possible instances of final game configurations, each reporting the status (blank, X, or O) of all 9 positions (identified by their horizontal and vertical position on the board). Thus, with respect to the previous dataset, here the complexity is augmented by the greater number of types, the greater number of values per type and the elimination of a significant portion of all possible board configurations (specifically, all non-final ones). In this case, $\overline{k} = 9$ (even if no new constraints of size 7, 8 and 9 were found), just like the number of types (each corresponding to one possible position) correctly recognized by the system:

1. *Top-Right position content*: {tr_b, tr_o, tr_x}
2. *Top-Center position content*: {tc_b, tc_o, tc_x}
3. *Top-Left position content*: {tl_b, tl_o, tl_x}
4. *Middle-Right position content*: {mr_b, mr_o, mr_x}
5. *Middle-Center position content*: {mc_b, mc_o, mc_x}
6. *Middle-Left position content*: {ml_b, ml_o, ml_x}
7. *Bottom-Right position content*: {br_b, br_o, br_x}
8. *Bottom-Center position content*: {bc_b, bc_o, bc_x}
9. *Bottom-Left position content*: {bl_b, bl_o, bl_x}

Lastly, the Congressional Votes [9] dataset describes 435 Congressmen as being democrats or republicans according to their votes on 16 issues. It is made up of 435 examples, described by means of 32 predicates, each representing the favorable (y) or opposite (n) vote on one of the above issues. It is particularly interesting because a certain amount of noise is present in the descriptions, in the form of unknown (omitted) votes, as reported in Table 1. Nevertheless, the algorithm is able to correctly infer all the 16 types (corresponding to the issues), each with its 2 descriptors (corresponding to the yes/no options).

Issue	No. of Omissions
handicapped infants	0
crime	25
adoption budget resolution	48
mx missile	15
physicians fee freeze	11
el salvador aid	11
religious groups in schools	15
immigration	22
synfuels corporation cutback	7
education spending	21
water project cost sharing	12
duty free exports	17
aid to nicaraguan contrast	14
superfund right to sue	31
export administration act S.A.	28
anti satellites test ban	11

Table 1. Noise on Congressmen votes

4 Experiments with Incomplete Knowledge

Once assessed the validity of the proposed algorithm in contexts with different characteristics, an interesting issue is evaluating its effectiveness under stress. A preliminary idea about this was given by the Congressional Votes dataset, where a number of observations missed some votes. Specifically, the aim was checking if it works also in presence of a small amount of information, and to what extent it does. To this purpose, we focused on the Scientific Papers dataset, for a number of reasons. First, because it is a real-world one, and is probably the most complex among those considered. Second, the shape of the descriptions is not fixed, differently from the Votes, Multiplexer and Tic Tac Toe ones. Third, it was made up of many different observations, differently from the Family one. Various experiments were run, in which noise was progressively introduced in the dataset descriptions. For each fixed amount of noise to be introduced, 10 random corruptions of the dataset were performed, on which running the proposed algorithm. Then, the learned types were checked and categorized in one of the following categories (listed by decreasing desirability): *correct*, *incomplete* (i.e., missing some types or some values in some type domains, but without mixing values belonging to different types), *impossible* (when the algorithm autonomously recognized that the available information was too loose for getting to a correct solution), and *wrong* (when at least one of the identified types contained in its domain values actually belonging to different types).

A first experiment in this direction aimed at assessing how sensitive the algorithm is to the amount of observations provided to it. In this case, the dataset corruption consisted in progressively eliminating observations (examples) from it (remember that the initial size was 112). The amount of corruption ranged

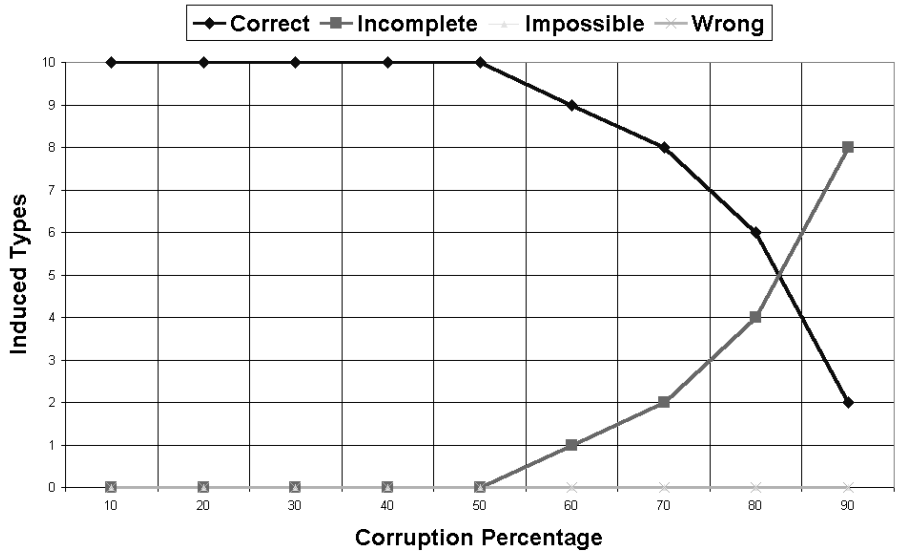

Fig. 1. Performance for Progressively Smaller Datasets

between 10% and 90% of the entire dataset, and the corresponding results are reported in Figure 1. It is interesting to note that the algorithm never generated undesirable (i.e., impossible or wrong) type domains. Actually, up to 50% of the dataset it always gave correct and complete answers. After that threshold, completeness started decreasing, but even when 90% of the observations was dropped (i.e., only 12 paper descriptions were available) in 2 cases it succeeded in finding the correct and complete types. This should allow one to state that the system is effective also when provided with very few observations.

Then, the next question was how much noise could be present in the available knowledge in order for the system not to be misleaded in its task. For this purpose, all the available observations were corrupted by eliminating from them a progressively larger amount of information, ranging from 10% to 60%. The experimental outcomes, graphically represented in Figure 2, suggest that the algorithm is more sensitive to partial descriptions than it was to a small number of observations. Indeed, in this case complete and correct types are induced only up to 20% of corruption, while accepting also incomplete types is ok up to 30%. Anyway, also after that threshold, the sum of desirable cases (i.e., correct and incomplete ones) far outperforms the number of undesirable ones. Only when 60% of each description in the dataset is dropped the number of wrong inductions becomes predominant, but interestingly it does not exceed half of the trials.

This behaviour can be explained because the proposed algorithm heavily relies on co-occurrence of values for inducing the type domains. Thus, eliminating whole observations, but leaving complete the remaining ones, potentially still

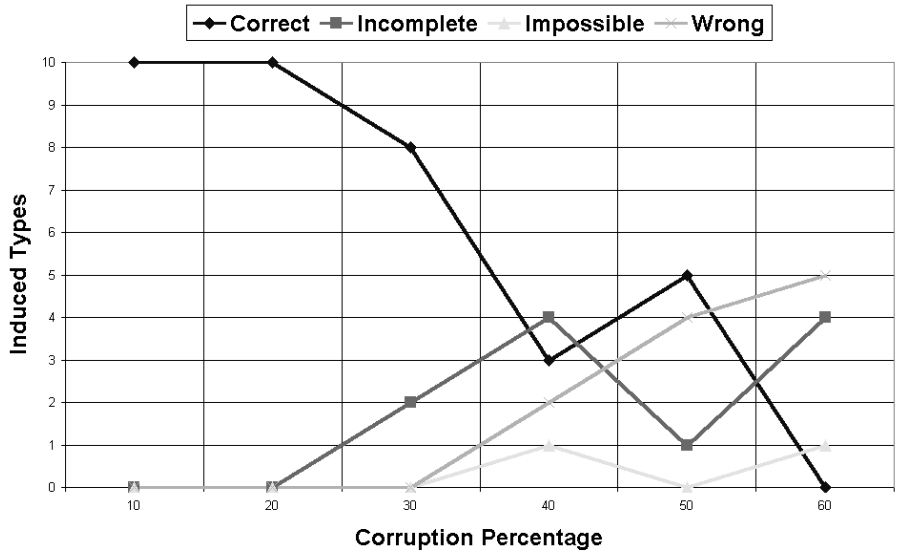

Fig. 2. Performance for Progressively Incomplete Descriptions

preserves many co-occurrences. On the contrary, dropping portions of each observation is likely to introduce false (supposed) incompatibilities among values that actually belong to different types. As already pointed out, some of these false incompatibilities are already present in the complete dataset (e.g., a line can have any width or height but is never too thick), thus artificially adding more noise of this kind makes an already hard task even harder. However, if the procedure is to be used in a Machine Learning context, incomplete (unknown) information in the available observations is a problem on its own, and experimental results show that abductive operators can cope with it only to some extent, which is in any case far below the threshold after which the proposed algorithm's performance becomes too low to be acceptable (and in general does not deal with datasets in which all descriptions are corrupted).

5 Conclusions

Many learning systems known in the literature are able to exploit and/or require knowledge about the types used in the description language and their related domains to improve their performance by pruning accordingly the search space of all possible hypotheses. This paper proposed an algorithm to automatically identify this kind of meta-information from the same observations that are input to the learning process, providing detailed examples of its behaviour. Experimental evaluation in several domains with characteristics that stress different features of the algorithm reveals encouraging performance. Moreover, being the algorithm

dependent on the amount and quality of observations available, specific experiments have been run aimed at assessing its robustness, even when incomplete information is provided.

Given the good performance of the algorithm by itself in identifying type domains from observations, the next step will be exploiting the induced meta-information to support the inductive step of learning algorithms, in order to assess the gain in computational effort and predictive accuracy that it can bring, in particular when the available descriptions are incomplete and abduction is to be used. Additional future work will concern a theoretical study of the algorithm behavior and complexity, in order to develop heuristics that can improve its performance by avoiding unnecessary computations. A comparison with other (e.g., Constraint Satisfaction Problem – CSP) solutions to the same task is also planned. Then, the next objective will be studying the case of structured types, which causes additional interrelations among descriptors to be taken into account.

Acknowledgement

This work was partially funded by the EU project IST-507173 Two Knowledge VIKEF, "Virtual Information and Knowledge Environment Framework".

References

[1] D. W. Aha. Incremental constructive induction: An instance-based approach. *Proceedings of the 8th International Workshop on Machine Learning*, pages 117–121. Morgan Kaufmann, 1991.

[2] H. Blockeel and L. De Raedt. Inductive database design. In *Foundations of Intelligent Systems*, volume 1079 of *Lecture Notes on Artificial Intelligence*, pages 376–385. Springer, 1996.

[3] R.M. Cameron-Jones and J.R. Quinlan. Efficient top-down induction of logic programs. *SIGART bulletin*, 5(1):33–42, 1994.

[4] L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26(2):99–146, 1997.

[5] W. V. de Velde. IDL, or Taming the Multiplexer Problem. *Proceedings of the 4th European Working Session on Learning*. Pittman, 1989.

[6] S. Ferilli, N. Di Mauro, T.M.A. Basile and F. Esposito. Incremental Induction of Rules for Document Image Understanding. In *AI*IA 2003: Advances in Artificial Intelligence*, volume 2829 of *Lecture Notes on Artificial Intelligence*, pages 176–188. Springer, 2003.

[7] P.A. Flach and N. Lachiche. Cooking up integrity constraints with primus. Preliminary Report CSTR-97-009, University of Bristol - Department of Computer Science, December 1997.

[8] P.A. Flach and N. Lachiche. Confirmation-guided discovery of first-order rules with `Tertius`. *Machine Learning*, 42(1/2):61–95, 2001.

[9] A. Kakas and F. Riguzzi. Abductive concept learning. *New Generation Computing*, 1999.

[10] E. McCreath and A. Sharma. Extraction of meta-knowledge to restrict the hypothesis space for ilp systems. In *Proceedings of the 8th Australian Joint Conference on Artificial Intelligence*, pages 78–82. World Scientific, 1995.

[11] R. S. Michalski. Inferential theory of learning. developing foundations for multistrategy learning. In R. S. Michalski and G. Tecuci, editors, *Machine Learning. A Multistrategy Approach*, volume IV, pages 3–61. Morgan Kaufmann, San Mateo, CA, U.S.A., 1994.

[12] K. Morik. Balanced cooperative modeling. *Machine Learning*, 11:217–235, 1993.

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

[14] E. Shapiro. Inductive inference of theories from facts. Technical Report 192, Computer Science Department, Yale University, 1981.