Learning with an Object-Oriented Data Model

M. Milano A. Omicini F. Riguzzi

LIA - DEIS, Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
Tel.: +39 51 6443086/6443087, Fax: +39 51 6443073
{mmilano, aomicini, friguzzi}@deis.unibo.it

1 The Problem

Inductive Logic Programming (ILP henceforth, [2]) is generally understood as a framework where general rules can be learned starting from a set of (positive and negative) examples and a background knowledge. While the background knowledge is typically a unique theory representing universal truths, learning intrinsically implies incompleteness of the information about the world. A multi-theory approach seems better suited for dealing with knowledge incompleteness since each theory intrinsically models a limited portion of the world.

Here, we aim at investigating the impact of an object-oriented data model on the learning process, where the background knowledge is structured as a collection of logic theories, organised in a hierarchy, and child/parent relationships between theories are reinterpreted as class/superclass relationships. Therefore, the background knowledge can be represented as a couple $\mathcal{P} = \langle T_{\mathcal{P}}, IsA_{\mathcal{P}} \rangle$ where $T_{\mathcal{P}}$ is a collection of labelled clauses, while $IsA_{\mathcal{P}}$ is a set of *parent* relations. A labelled clause has the form $Theory\ :\ Head :\!- Body$, representing a clause $Head :\!- Body$ of the theory denoted by the ground term $Theory$. In this perspective, the class tree can be exploited in the generalisation phase by learning a concept at different levels of granularity.

In our multi-theory environment, both the background and the learned theory are multi-theory logic programs, and the learning problem can be defined as follows:

Given: a program $\mathcal{P} = \langle T_{\mathcal{P}}, IsA_{\mathcal{P}} \rangle$ (*background knowledge*)
a set E^+ of positive examples defined on instances
a set E^- of negative examples defined on instances
Find: a program $\mathcal{P}' = \langle T'_{\mathcal{P}}, IsA'_{\mathcal{P}} \rangle$ such that
$\forall e^+ \in E^+,\ \mathcal{P}' \models_M e^+$
$\forall e^- \in E^-,\ \mathcal{P}' \not\models_M e^-$.

where $\models_M$ represents the notion of entailment in a multi-theory environment where overriding is chosen as the default mechanism for theory composition [3]. In this problem, we choose to take as fixed the class hierarchy and to learn no new *parent* relations, so that $IsA_{\mathcal{P}} = IsA'_{\mathcal{P}}$. (An interesting generalisation would instead consider this form of learning, too. Further work will be devoted to it.) Positive and negative examples are provided as

atomic formulae which are true or false w.r.t. a given theory, i.e., they represent specific instance properties that should be generalised by the learning process.

2 Sketching an Algorithm

There are many choices to be made when designing a learning algorithm for this problem. Here, we briefly sketch a possible algorithm we implemented in SICStus Prolog [5], and illustrate it through some simple examples.

The algorithm basically works in two steps: learning on instances and learning on classes. In each instance, we first computes the least general generalization of the atoms representing positive examples using a notion of least general generalisation, denoted with τ-*lgg*, which properly extends Plotkin's *lgg* [4] in order to cope with hierarchical types. Examples below will give an intuition of how τ-*lgg* works. The resulting clause is then tested for consistency wrt negative examples and, in case it is found inconsistent, it is specialized by iteratively adding a literal to it.

After the first step, each instance is correctly represented by a structured theory which entails all the positive examples and does not entail the negative ones. Therefore, we can start the recursive learning process on classes from final classes up to the class hierarchy root. In short, for every class theory F, we try to generalise clauses with the same predicate symbol and arity occurring in the child theories of F by computing their τ-*lgg* and adding it to F. Thanks to the adoption of overriding as the default theory composition mechanism, we no longer need to consider positive and negative examples defined on instances when learning in classes.

As an example of how the algorithm sketched above works, consider a multi-theory background knowledge $\mathcal{P} = \langle T_{\mathcal{P}}, IsA_{\mathcal{P}} \rangle$ such that

$$\begin{aligned} IsA_{\mathcal{P}} &= \{ \; dog \Vdash animal, cat \Vdash animal, fish \Vdash animal, \\ &\quad kitty \Vdash cat, fufy \Vdash cat, toby \Vdash dog, buck \Vdash dog, \\ &\quad wanda \Vdash fish, peggy \Vdash fish, sally \Vdash fish, \\ &\quad kitkat \Vdash dog\_food, gourmetdog \Vdash dog\_food, doggy \Vdash dog\_food \\ T_{\mathcal{P}} &= \{ \; buck{:}likes(kitkat), buck{:}likes(doggy), buck{:}likes(gourmetdog), \\ &\quad toby{:}likes(gourmetdog), toby{:}likes(kitkat) \; \} \end{aligned}$$

so that all theories of $\mathcal{P}$ are empty except for *buck* and *toby*. Consider now the following positive examples:

$$E^+ = \{|lives\_in(water)|_{sally}, |lives\_in(water)|_{wanda}\}$$

As a first step, we perform a quite trivial form of learning with respect to (theories representing) instances. Since we have only one positive example for each instance and no negative examples, the examples are directly added to the two fish instances in the form of the following clauses

$$sally{:}lives\_in(water) \qquad wanda{:}lives\_in(water)$$

As the second step, we consider the two positive examples representing a property common to all fish (more precisely, a property common to all fish for which something is known: *peggy* is a fish, but nothing is known about where she lives). Then, we learn that every fish lives in water, by learning the fact $fish{:}lives\_in(water)$, which is trivially the τ-*lgg*

of the two clauses for predicate *lives_in/1* in *wanda* and *sally*. Since now it holds as well that

$$\mathcal{P}' \models_M |lives\_in(water)|_{peggy}$$

we come to learn something about where *peggy* lives, too. In fact, the inheritance rules (through the theory composition mechanisms adopted) ensure that every instance of class *fish* having no clauses for *lives_in/1* (like in the case of *peggy*) inherits the one from its parent theory *fish*. Finally, it is worth noting that the same inheritance mechanisms potentially allows us to discard clauses $sally{:}lives\_in(water)$ and $wanda{:}lives\_in(water)$.

Consider now the following examples for the cat instances *kitty* and *fufy*:

$$E^+ = \{|wants\_to\_eat(wanda)|_{kitty}, |wants\_to\_eat(peggy)|_{kitty}, |wants\_to\_eat(peggy)|_{fufy}\}$$

In order to perform the first step of the learning process, that is, learning in the single instance theory, we adopt a *typed* ILP algorithm. This algorithm learns the same rules as an ILP algorithm [1], but exploits the class hierarchy for the parameter generalisation by computing the least upper bound on each parameter. For example, we can learn the following rule for instance *kitty*:

$$kitty \ : \ wants\_to\_eat(X) :- X \models\!\prec fish$$

while for instance *fufy* we just put the fact $wants\_to\_eat(peggy)$ in the associated theory.

Now the system generates a rule for cats that is the τ-*lgg* of the two clauses learned wrt its instances. In particular, the system generates the rule:

$$cats \ : \ wants\_to\_eat(X) :- X \models\!\prec fish$$

since the clause for *kitty* is simply a generalisation of the clause for *fufy* with respect to the background knowledge $\mathcal{P}$. It is worth noting that here we learn a meaningful non-ground rule whose body contains no other predicates than the hierarchic one, actually classifying the argument of the clause head.

Like in ILP, the presence of negative examples may require a specialisation step exploiting the clausal part of the background knowledge, according to the specific language bias. Then, consider the following set of positive and negative examples:

$$\begin{aligned} E^+ &= \{ \ |eats(kitkat)|_{toby}, |eats(gourmetdog)|_{toby}, \\ &\quad\ |eats(kitkat)|_{buck}, |eats(gourmetdog)|_{buck}, |eats(doggy)|_{buck} \ \} \\ E^- &= \{ \ |eats(doggy)|_{toby} \ \} \end{aligned}$$

Since there are only positive examples for instance *buck*, we learn the clause

$$buck \ : \ eats(X) :- X \models\!\prec dog\_food.$$

by exploiting the hierarchy and generalising on parameters in the same way as shown in the previous examples.

Instead, as far instance *toby* is concerned, the clause $toby \ : \ eats(X) :- X \models\!\prec dog\_food.$ covers also the negative example, so it has to be specialised according to the language bias. Suppose that predicate *likes/1* can be used for clause specialisation, we add a literal to the clause body obtaining the clause

$$toby \ : \ eats(X) :- X \models\!\prec dog\_food, likes(X).$$

which covers all positive examples and does not entails the negative one.

In the next step, the generalisation process takes the least general τ-generalisation

$$dog \ : \ eats(X) :- X \mid\!\prec dog\_food.$$

of the clauses previously learned for the two instances, and adds it to their superclass *dog*. It is worth noting that in this case the presence of the correct clause for the theory *toby* combined with the overriding composition policy prevents us from generating incorrect programs, even though negative examples have not been taken into account in the generalisation process performed with respect to classes.

3 Discussion

Generally speaking, adopting an object-oriented data model in the representation language gives us a number of advantages over ILP. First of all, it increases the expressiveness of both background and target theories. Knowledge related to a given domain abstraction is naturally encapsulated into a single theory, which is quite useful when highly complex and structured domains are to be handled. Even more, while ILP does not syntactically support the distinction between classes and properties, which are all represented by means of predicate symbols, our approach provides a structural support for the notion of classes, which are logic theories, denoted by terms, while properties are still represented as predicates.

Then, the class hierarchy can be exploited so as to guide the learning process. The hierarchical structure of a multi-theory program suggests a finer granularity for the learning process: different clauses for the same property related to different classes of the same hierarchy branch represent the result of an incremental learning process with a growing degree of generalisation. For example, in the class *dog*, we learn a definition for *eats* that is more general than the one learned for *buck* and *toby*.

This can be combined with the exploitation of the inheritance mechanisms: for instance, adopting overriding as the default rule for theory composition protects the learning process from inconsistency.

Finally, the partition of the domain representation in a multiplicity of logic theories increases the efficiency of the testing of the clauses against examples. In fact, the derivation of an example is done using only clauses in the context representing the involved instance. This seems a particularly relevant feature since it has been shown that the coverage test of a clause is the most time-consuming task of ILP algorithms.

References

[1] F. Bergadano and D. Gunetti. *Inductive Logic Programming.* MIT Press, 1996.

[2] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295 317, 1991.

[3] A. Omicini. A general framework for multi-theory logic languages. Technical Report DEIS-LIA-009-96, University of Bologna (Italy), 1996. LIA Series no. 16.

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

[5] Swedish Institute of Computer Science, Kista, Sweden. *SICStus Prolog User's Manual.*