IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

Learning Relations and Logic Programs

This is the author's manuscript

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/123096 since

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

29 March 2024

The Knowledge Engineering Review, Vol. 7:1, 1994, 73–77

Learning relations and logic programs

F. BERGADANO* and D. GUNETTI†
**Dipartimento di Matematica, University of Catania, Italy*
†Dipartimento di Informatica, University of Torino, Italy

1 Introduction

Inductive Logic Programming (ILP) is an emerging research area at the intersection of machine learning, logic programming and software engineering. The first workshop on this topic was held in 1991 in Portugal (Muggleton, 1991). Subsequently, there was a workshop tied to the Future Generation Computer System Conference in Japan in 1992, and a third one in Bled, Slovenia, in April 1993 (Muggleton, 1993). Ideas related to ILP are also appearing in major AI and machine learning conferences and journals. Although European-based and mainly sponsored by ESPRIT, ILP aims at becoming equally represented elsewhere; for example, among researchers in America who are investigating relational learning and first order theory revision (see, for example, the papers in Birnbaum and Collins, 1991) and within the computational learning theory community. This year's IJCAI workshop on ILP is a first step in this direction, and includes recent work with a broader range of perspectives and techniques.

Many different problem settings have been proposed, but it is still possible to abstract similarities and identify one simplified ILP problem:

Given:
A hypothesis space H
A set of Positive Examples E+
A set of Negative Examples E−
A body of prior knowledge K

Find:
A hypothesis $h \in H$ such that
(1) $K,h \models e+$, for all $e+ \in E+$ (completeness)
(2) $K,h \models e-$, for all $e- \in E-$ (consistency)

where, in the majority of approaches,

- H is a set of possible logic programs, often defined by means of a set C of possible Horn clauses; in this case $H=2^C$. For instance, C may be defined as all clauses having ancestor(X,Y) as their head, and a body formed with any conjunction of literals obtained with the predicate symbols "parent" and "ancestor".
- Positive and Negative examples are ground atoms, e.g., ancestor(paul, susan).
- The prior knowledge K is often just a Horn theory, e.g.,

```
parent(X,Y):- mother(X,Y).
parent(X,Y):- father(X,Y).
father(paul,jane).
mother(jane,susan).
```

- The desired hypothesis is a logic program $P \subseteq C$, e.g.,

```
ancestor(X,Y):- parent(X,Y).
ancestor(X,Y):- parent(X,Z), ancestor(X,Y).
```

The motivation for the above problem setting is twofold:

- From a machine learning point of view, this is the natural evolution of work in concept acquisition and relational learning during the past ten years, where researchers have moved towards more and more expressive representation formalisms: from numerical classifiers, to propositional formulas and decision trees, then to descriptions containing variables with

restricted forms of quantification and non-recursive Horn clauses. This evolution has allowed for more complex classification applications, where the mutual relationships among object components is essential for distinguishing positive and negative examples.

- For logic programming and software engineering, this represents a body of novel techniques for program synthesis from incomplete specifications. Logic programs are particularly adequate for inductive synthesis, because of their simple syntax, and because it is often possible to learn individual clauses (= small-scale subprograms) one at a time and independently of each other. Moreover, as pure logic programs have a declarative meaning, an incomplete specification by examples makes perfect sense. There are strong ties between ILP and work done on Logic Program Synthesis and Transformation, and Dr. K K Lau, who has chaired two previous international workshops on this topic (Lau & Clement, 1993), is an invited speaker at our meeting. For software engineering, there are also applications in the area of fault-based program testing (Bergadano, 1993).

2 Background

Early work on issues now developed within the ILP area are well represented by Plotkin's study on least general generalizations (Plotkin, 1970) and Shapiro's Model Inference System (Shapiro, 1983). The former is related to the problem of generalizing clauses, as a basis for the bottom-up induction of logic formulas: the least general generalization of a number of examples will serve as a compressed description. The latter is based on the top-down specialization of clauses, until a logic program which is consistent with the available examples is produced.

Plotkin was the first to rigorously analyse generalization, based on the notion of θ-subsumption. He did not restrict himself to Horn clause logic; in fact logic programming did not yet exist at that time.

A clause C θ-subsumes (or is a generalization of) a clause D, if there exists a substitution θ such that $C\theta \subseteq D$. Usually this is written $C \preccurlyeq D$. We say that C is the least general generalization (lgg) of D if $C \preccurlyeq D$ and, for every other E such that $E \preccurlyeq D$ it is also the case that $E \preccurlyeq C$. For example (Plotkin, 1970), consider the following two ground clauses:

C_1 = win(conf1):-occ(place1,x,conf1),occ(place2,o,conf1).
C_2 = win(conf2):-occ(place1,x,conf2),occ(place2,x,conf2).

which represent two winning configurations in a two-person game, where two board positions can be occupied by either an 'x' or an 'o'. The lgg of the two clauses, after the elimination of redundant literals, turns out to be:

$lgg(C_1,C_2)$ = win(Conf):-occ(place1,x,Conf),occ(place2,Y,Conf).

Plotkin's ideas on least general generalizations are the basis for subsequent bottom-up ILP algorithms: the solution program P will be computed as the lgg of the given positive examples, and we move from most specific formulas (the examples) towards more general descriptions, until a consistent solution is output. Plotkin also introduces a notion of relative lgg to take into account the given prior knowledge K. Recent bottom-up inductive methods such as Golem (Muggleton, 1991) and CLINT (DeRaedt, 1991) are based on this scheme.

Shapiro's Model Inference System (MIS) is concerned with the inductive synthesis of logic programs. The method is incremental, in the sense that it maintains a current inductive hypothesis (a logic program P), and modifies it as new examples are provided. By contrast, one step methods require all the examples at the beginning, and try to generate a program which is consistent with those examples.

The hypothesized program P can be modified in two ways: (1) $P \vdash a$, but a is false, and therefore P must be weakened; and (2) $P \not\vdash a$, but a is true and therefore P must be made stronger. P can be weakened by either removing one of its clauses (as done in MIS) or by making a clause weaker, e.g., by adding literals to its body. P can be strengthened by either adding a new clause (as done in MIS) or by making one of its clauses more general, e.g., by removing literals from its body. MIS keeps on doing modifications of type (1) or (2) until P is consistent with all of the examples currently available. After this, it will be ready to read a new example.

Strengthening P is done by "refining" some previously removed clause. Refining is a top-down specialization process where new literals are added one at a time to the antecedent of the selected clause. We then move from very general clauses, that will cover both positive and negative examples, towards more specific ones, until a set P of consistent clauses that cover all positive examples is found. This scheme is also found in recent top-down ILP systems, including ML-SMART (Bergadano & Giordana, 1988), Foil (Quinlan, 1990) and Focl (Pazzani & Kibler, 1992).

3 Main themes of the meeting

The ILP problem can be hard to solve. This may be due to the number and the complexity of the examples, or even to the efficiency of the background knowledge, but the main reason is the size of the hypothesis space H. As a consequence, all practical induction algorithms need some prior information or some heuristics for searching for a solution in H. This prior information is often called an inductive bias, and can take different forms, depending on the nature of the learning algorithm and on the desired collaboration from the user.

Rouveirol, Ade and De Raedt (1993) analyse the kinds of inductive bias that are relevant in bottom-up ILP systems. Their analysis is based on the observations that different approaches to bottom-up induction of relational concepts can be mapped to the following algorithmic framework:

```
Completed_Examples := ∅
for all positive examples e+ do
        SC := starting_clauses(e+,K,B)
        Completed_Examples := Completed_Examples ∪ SC
generalize (e.g. via lgg) Completed_Examples
```

where K is a theory playing the role of a body of background knowledge, and B is an inductive bias. A starting clause C for an example e+ is obtained incrementally: at each stage a literal l taken from a set S is added to C, until no other literals can be added without making C inconsistent. The choice of the set S of possible literals is a form of static pruning determined by the bias B. Moreover, a literal $l \in S$ will be added to the clause C, only if a kind of dynamic pruning condition holds, based on the bias B and the particular example e+ that is being considered.

The systems CLINT (DeRaedt, 1991), GOLEM (Muggleton, 1991) and ITOU (Rouvierol, 1994) can be instantiated to the above scheme:

- CLINT: S contains all literals that are syntactically acceptable in the chosen representation language. A literal l is dynamically allowed if it is true w.r.t. the background theory and $sc \wedge l$ covers e+.
- GOLEM: S contains all ground literals that are true w.r.t. the background theory. A literal is statically allowed if it has a depth bound of i. It is dynamically allowed if it is j-determinate.
- ITOU: A literal is added to SC if it follows from the theory T, the example e+ and the previously added literals in sc.

Cohen (1993) compares different kinds of bias schemes using the top-down Grendel system; these include ij-determinism, k-free clauses, k-local clauses and a bias based on relational pathfinding. The latter was implemented in a special case by means of a small set of rule schemata.

All of the above forms of bias were translated into antecedent description grammars (ADGs). Grendel, with an appropriate ADG, works as a prototype for an ILP system using that bias, and allows the user to do some experimentation with different kinds of biases. Results show that (1) ij-determinism is best for simple Prolog programs, e.g., append; (2) k-locality is good for the "loan" database; (3) relational pathfinding is good for the "family" database.

Bias is important for both bottom-up and top-down systems. However, if learning is to be fully automated, one could reasonably ask where the bias comes from. One possible answer, which is often well-motivated, is that the bias must come from the user, because if there is no prior information of this kind, learning is inaccurate and inefficient. Another answer lies in the possibility of learning the bias from the environment.

To this aim, Silverstein and Pazzani (1993) propose a technique for learning a form of preference bias based on relational cliches. These indicate conjunctions of literals which are likely to be useful

in a Foil-like top-down learner. Such conjunctions will then be added in a single specialization step, instead of one literal at a time. If many cliches are given to the system, learning may turn to be slower, as the branching factor in the specialization tree is increased. However, particular conjunctions may be useful in two cases:

- they often occur in the solution clauses; then the increased branching factor is remedied by the smaller number of specializations needed to reach some consistent rule;
- the conjunction as a whole produces some important information gain, while the component literals do not. Then the heuristics used by Foil-like algorithms might hide away an existing solution. Relational cliches seem to be a simpler case of preference biases expressed by domain theories as in FOCL or by ADGs as in Grendel.

Silverstein and Pazzani (1993) suggest a method for learning cliches. The idea is simple:

- use very general cliches on a number of simple domains;
- specialize those cliches to their most useful instantiations;
- use the specialized cliches in more complex domains.

In general, learning a bias is a controversial objective, because bias is a precondition of learning, which must be given. However, the above framework is meaningful: a weak bias is used to learn simple concepts, and is made stronger on the basis of the obtained solutions. The specialized, stronger bias is then used on more difficult problems. All this obviously rests on the assumption that the complex problem is in some way related to the simpler domains.

As another form of bias, one could consider an initial program P, which may be seen as a tentative solution given by the user. This is meaningful if the correct program is close to this initial guess, e.g., if it can be obtained by adding or deleting a small number of clauses or literals. In this case, the ILP problem is transformed into what is called a theory revision task.

Wogulis (1993) considers the problem of revising logic programs with negation, and presents a system, called A3, together with some experiments. The experiments suggest that A3 produces an accuracy which is comparable to FOCL, but yields a result which is closer to the theory given initially.

Previous approaches to theory revision follow two major schemes:

- operationalization (e.g., ML-SMART (Bergadano & Giordana, 1988), FOCL (Pazzani & Kibler, 1992), Grendel (Cohen, 1993)); the domain theory is unfolded on the basis of given examples, until more specific, operational descriptions are obtained. These descriptions may then be specialized further using standard inductive techniques. This idea cannot work when the original theory contains negation, because the operationalization cannot be pushed into a negated literal, which is interpreted via negation as failure.
- modification (e.g., AUDREY (Wogulis, 1993), Forte (Richards & Mooney, 1991)); if a positive example is not covered the theory is generalized, if a negative example is covered, the theory is specialized. If negation is present, the above scheme is inappropriate. For instance, a negative example may be covered because the call of a negated literal in its definition fails; in that case the definition of that literal should be generalized.

In the end, even if there is some strong bias which is given by the user, learning will be efficient only if the hypothesis space H is sufficiently constrained. Systems such as LOPSTER (Aha et al., 1993) require that the program to be learned be extremely simple, i.e., programs containing one non-recursive base clause and one linear recursive clause. In general, it is not essential that the solution program P be simple; the important datum is the size of the hypothesis space H. If P is a complex program, but H only contains P and a few alternatives, then learning may still be feasible and likely to be successful.

4 Conclusion and projections

The conclusion is based on the previous observation. ILP has the potential of learning quite complex programs, if sufficient prior information is given, for example by means of a restricted hypothesis space H that still contains a correct solution. Such information must be given by the

user, but in a software engineering setting this may be acceptable: the user is a programmer who knows a lot about the program which is being developed. However, not all this knowledge is written in general rules or procedures; part of it is factual and goes into positive and negative examples. ILP will provide tools for turning such diverse sources of information into an executable procedure, which may then be refined further by existing program transformation techniques.

References

Aha, D, Ling, C, Matwin, S and Lapointe S, 1993. "Learning singly recursive relations from small datasets". In: *Proceedings IJCAI-93 Workshop on ILP*, pp 47–58.

Bergadano, F, 1993. "Test case generation by means of learning techniques". In: *Proceedings ACM SIGSOFT*. Los Angeles, CA.

Bergadano, F and Giordana, A, 1988. "A knowledge intensive approach to concept induction". In: J. Laird (ed.), *Proceedings of the Fifth International Conference on Machine Learning*, pp. 305–317. Ann Arbor, MI: Morgan Kaufmann.

Birnbaum, L and Collins, G (eds.), 1991. *Proceedings 8th International Conference on Machine Learning, Part VI: Learning Relations*. Morgan Kaufmann.

Cohen, W, 1993. "Rapid prototyping of ILP systems using explicit bias". In: *Proceedings IJCAI-93 Workshop on ILP*, pp 24–35.

DeRaedt, L, 1991. *Interactive Concept Learning*. PhD, Thesis, Katholieke University, Leuven, Belgium.

Muggleton, S (ed.), 1991. *Inductive Logic Programming*. Academic Press.

Muggleton, S (ed.), 1993. *Proceedings Third International Workshop on Inductive Logic Programming*. Bled, Slovenia.

Pazzani, M and Kibler, D, 1970. "The utility of knowledge in inductive learning". *Machine Learning* 9 57–94.

Plotkin, G, 1970. "A note on inductive generalization". In: B Meltzer and D Michie (eds.), *Machine Intelligence 5*, pp 153–163.

Quinlan, R, 1990. "Learning logical definitions from relations". *Machine Learning* 5 239–266.

Richards, BL and Mooney, LJ, 1991. "First-order theory revision". In: *Proceedings International Conference on Machine Learning*, pp 447–451. Evanston, IL.

Rouveirol, C, 1993. "Flattening: a representation change for generalization". *Machine Learning* Special issue on Evaluating and Changing Representation, K Morik, F Bergadano and W Buntine (eds.).

Rouveirol, C, Ade, H and DeRaedt, L, 1993. "Bottom-up Generalization in ILP". In: *Proceedings IJCAI-93 Workshop on ILP*, pp 59–70.

Shapiro, EY, 1983. *Algorithmic Program Debugging*. MIT Press.

Silverstein, G and Pazzani, M, 1993. "Learning relational cliches". In: *Proceedings IJCAI-93 Workshop on ILP*, pp 71–82.

Wogulis, J, 1993. "Handling negation in first order theory revision". In: *Proceedings IJCAI-93 Workshop on ILP*, pp 36–46.