Intensional Learning of Logic Programs

D. Gunetti and U. Trincherò

Università di Torino, Dip. di Informatica
Corso Svizzera 185, 10149 Torino, Italy
{gunetti,trincher}@di.unito.it

Abstract

In this paper we investigate the possibility of learning logic programs by using an intensional evaluation of clauses. Unlike learning methods based on extensionality, by adopting an intensional evaluation of clauses the learning algorithm presented in this paper is correct and sufficient and does not depend on the kind of examples provided. Since searching a space of possible programs (instead of a space of independent clauses) is unfeasible, only partial programs containing clauses successfully used to derive at least one positive example are taken into consideration. Since clauses are not learned independently of each others, backtracking may be required.

1 Introduction

Inductive Logic Programming (ILP) is the field of Machine Learning concerned with the problem of learning logic programs from ground examples of their input-output behavior.

Since learning logic programs is in general hard, many systems (such as Foil [2]) try to achieve efficiency by adopting an *extensional* evaluation of candidate clauses. However, since clauses are learned extensionally, but then the whole program is interpreted intensionally (i.e. it is run on a Prolog interpreter), extensional methods are, in general, not correct nor sufficient¹[1], unless some (or even many) special examples are given to the system. Since all these examples must be covered, this can strongly slow the learning task.

The above problems are particularly serious if we want propose ILP also as a tool for Logic Programming and for Software Engineering at large. Especially in this case, the potential ILP user should be able to get a logic program each time it exists in the designed hypothesis space (and not only sometimes or even often), and the synthesized program should be correct (and not only “approximately” correct) at least on the seen examples. Moreover, the user should not be compelled to provide

¹ A program is *complete* if it derives all the positive examples, and it is *consistent* if it does not derive any of the negative examples. A learning system is *correct* if it only outputs complete and consistent programs, and it is *sufficient* if it finds a complete and consistent program if it exists in the hypothesis space.

an extensionally complete list of examples, since otherwise it could be simpler and/or faster to write down directly the required program.

In this paper we wish to investigate the possibility of fulfilling the above requirements with an alternative approach to the ILP problem. We present an ILP algorithm (actually, two slightly different versions of the same algorithm) which is correct and sufficient. The originality of the approach is that it adopts an intensional evaluation of candidate clauses. That is, clauses are checked against positive and negative examples by running them with a Prolog Interpreter. In this way the problems of extensionality are automatically overcome, since a logic program is learned in the same way as it will be used. The learning algorithm is automatically suitable for multiple predicate learning and it can work with any number of positive and negative examples. In particular, in no sense it requires an extensionally complete set of positive examples.

2 The Learning Algorithm

The induction procedure described in this section is called *Intensional Learner* (IL for short). First, given a set of clauses S and an example e such that $S \vdash_{SLD} e$, let us define a clause of S as *successful* (w.r.t. e) if it used in the proof of e . IL works as follows. Candidate clauses (CC) are produced from a description of the hypothesis space (HS), until a set of successful clauses (a partial program) deriving some positive example e^+ is found. The partial program p is added to the partial programs discovered previously, and the covered examples are removed, until no more positive examples remain. At every step, the whole set of clauses learned up to that point is checked against the negative examples, and if some of them are derived the learning task backtracks to a different derivation for e^+ . If no consistent set of clauses can be found in CC, a new clause from the hypothesis space is generated and added to CC, and a different possible solution is sought. Here is an informal description of IL (algorithm 1):

```

input: a set of positive and negative examples  $E^+$  and  $E^-$ 
       a description of the hypothesis space HS
       a background knowledge BK
CC  $\leftarrow \emptyset$ ; P  $\leftarrow \emptyset$ 
while  $E^+ \neq \emptyset$  do
   $c \leftarrow \text{generate\_one\_clause}(\text{HS})$ 
  CC  $\leftarrow \text{CC} \cup c$ 
  for each  $e^+$  such that  $\text{CC} \cup \text{BK} \vdash_{SLD} e^+$  do
    let  $p \subseteq \text{CC}$  be the set of clauses successfully used to derive  $e^+$ 
    P  $\leftarrow P \cup p$ 
    if  $\exists e^-$  such that  $P \cup \text{BK} \vdash_{SLD} e^-$  then backtrack
   $E^+ \leftarrow E^+ - e^+$ 

```

We note that each p is learned by simply running $\text{CC} \cup \text{BK}$ as a normal logic program on a Prolog interpreter on one of the positive examples, and maintaining a trace of

the clauses effectively used in the derivation of that example. In the same way the current P is checked against the negative examples.

We immediately notice that there is a major drawback in the above algorithm. If a set of partial programs consistent with the given examples cannot be found in the current CC , a new clause is added to CC from the hypothesis space, and the search starts again. However, a lot of time can be wasted by learning again the same partial programs as before (i.e. those not involving the new clause) and found inconsistent. A solution to this problem is to search from the very beginning the entire hypothesis space, i.e. by immediately assigning to CC all the clauses in HS (let's call this version of Intensional Learner as algorithm 2). Clearly, a serious drawback of this solution is that the hypothesis space must be finite, and it must be completely generated before the beginning of the learning task.

Since we can also define a hypothesis space containing clauses with different consequents, multiple predicate learning is automatically achieved. We note that both for making the learning procedure terminate, and to guarantee the termination of learned programs, we must require that candidate recursive clauses satisfy some well-ordering relation, such as in [2].

3 Discussion and Conclusion

We begin this section by addressing some major issues about the properties of the induction procedure.

First, if we suppose that the set of generated clauses CC forms a terminating program, then the following property holds:

Theorem 1: *IL is correct and sufficient.*

Proof: Trivial, as it is a consequence of the definition of the induction procedure. Obviously, in the case of algorithm 2 we must assume HS to be finite.

A second major point is about the computational complexity of the learning task. Extensional methods explore a hypothesis space HS of independent clauses, and hence they have a complexity which is linear in $|HS|$. On the contrary, an exhaustive search in the space of possible programs (i.e. pick up a subset of HS and check it against the given examples) would be exponential in $|HS|$, and practically unfeasible. Our approach stands between these two, as it only takes into consideration (partial) programs which derive at least one positive example, and where each clause is successful. To estimate the computational complexity of intensional learner (algorithm 1), consider a positive example e^+ such that $CC \cup BK \vdash_{SLD} e^+$ for some set CC . Let the *depth* of e^+ be the maximum number of clauses used in its derivation². For practical program induction tasks, it is often the case that the depth of an example is related to its complexity, and not to the set of candidate clauses CC . As a consequence, if we have n positive examples, d is the maximum depth on all the examples, S is the size of the first subset of the hypothesis space containing a

²For example, if we are working with sets or lists, and we can guarantee that all recursive calls in $CC \cup BK$ are on a smaller set or on a shorter list, then the depth of an example is the size of the sets or lists it contains.

complete and consistent program w.r.t. the examples, and we can assure that each partial program is discovered only once, the complexity of our method will be of the order of $S^{n \cdot d}$. For algorithm 2, since here a partial program can be derived at most once, we have a computational complexity of the learning task of the order of $[HS]^{n \cdot d}$ (to which we must add the time required to generate the hypothesis space).

Third, it is clear that the complexity of the learning task can be improved by controlling the number and kind of given examples. Unlike in extensional systems, in IL there is not *any* relationship between the given examples and the possibility to learn a program complete and consistent w.r.t. those examples. Simply, if such a program exists in the hypothesis space, it is found. This means that with an intensional evaluation of clauses, a positive example is sufficient to learn all the clauses necessary to derive it. By limiting the number (n) and the depth (d) of the positive examples, it is possible to improve the performance of the system without affecting its ability to learn.

We conclude by observing that the induction procedure we have presented is rather simple and expectable, and it could be improved in many ways. Nevertheless, even as it is, it shows that learning logic programs by adopting an intensional evaluation of clauses is feasible if 1) we avoid searching a space of possible programs and limit ourselves to a set of successful clauses and 2) we limit the size of the hypothesis space by some form of prior information and strong constraints. Even if the intensional approach is, in general, less efficient if compared with the extensional approach, it seems particularly suitable for software engineering applications, where the requirements for complete and consistent programs, and the ability to find them whenever they exist in the hypothesis space are a primary demand. Moreover, an intensional system can work with very few and simple examples of the target concept(s). This is important to limit the computational cost of the system, and it is fundamental if no more examples are available. We believe that a careful choice of the examples, together with a judicious design of the hypothesis space can make the efficiency of intensional systems comparable with the one of extensional systems.

Acknowledgement: This work was in part supported by BRA ESPRIT project 6020 on Inductive Logic Programming.

References

- [1] F. Bergadano. Inductive database relations. *IEEE Trans. on Data and Knowledge Engineering*, 5(6), 1993.
- [2] R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5:239-266, 1990.