Learning Recursion with Iterative Bootstrap Induction
(Extended Abstract)

Alípio Jorge and Pavel Brazdil

LIACC, University of Porto, Rua do Campo Alegre, 823, 4150 PORTO, Portugal
Tel. +351 2600 1672. Fax. +351 2600 3654. Email: {amjorge,pbrazdil}@ncc.up.pt

Abstract. In this paper we are concerned with the problem of inducing recursive Horn clauses from small sets of training examples. The method of iterative bootstrap induction is presented. In the first step, the system generates simple clauses, which can be regarded as properties of the required definition. Properties represent generalizations of the positive examples, simulating the effect of having larger number of examples. Properties are used subsequently to induce the required recursive definitions. This paper describes the method together with a series of experiments. The results support the thesis that iterative bootstrap induction is indeed an effective technique that could be of general use in ILP.

1. Introduction

One potential usage of ILP systems is in algorithm synthesis. However most ILP systems still require relatively large example sets which is rather impractical. Several people have proposed a solution (e.g. [1], [4], [6]). The solution described in [1] and incorporated in CRUSTACEAN exploits common substructures in the examples. Although encouraging results have been achieved, the method seems to be difficult to extend and integrate into a general purpose inductive system.

In this paper we investigate another method called *iterative bootstrap induction*, which represents an alternative approach to this problem. This method can be seen as a special case of the closed-loop learning strategy [5]. An implementation of this method proved experimentally to be able to induce recursive definitions from small sets of positive examples even if these were generated at random.

The system of Zelle et al. [7], CHILLIN, also employs the idea of closed loop learning to overcome incomplete example sets. However, CHILLIN uses a FOIL-like heuristic to guide the search and still needs relatively large example sets.

System SKIL

We adapted the system SKIL [2], which performs induction by refining algorithm sketches, to incorporate the iterative bootstrap induction method. SKIL takes as input positive and negative examples and optionally algorithm sketches. Background predicates are defined either extensionally or intensionally. The concept language is described by means of a definite clause grammar (DCG) similar to [3].

SKIL is a top-down covering system. For each positive example, SKIL looks for a sequence of ground facts that obtain the output arguments from the inputs. Each of those facts needs to either be proved from the background knowledge, or represent a positive example. Each sequence can give rise to a different candidate clause which is accepted as long as it does not cover negative examples. This process is similar to relational pathfinding described in [8].

2. Iterative Bootstrap Induction

The method presented constructs theories in a stepwise manner. In each step a tentative theory is produced which can be reused in the next cycle of the induction process. If certain stopping criteria are satisfied, the process terminates.

Let us see an example. Suppose the aim is to induce a definition of member on the basis of two positive examples (see Table 1) plus appropriate negative examples and background knowledge. If these are presented to SKIL it generates the theory T1. The two clauses generalize the positive examples and express in fact properties of the member relation. They are valid for the examples without necessarily being part of the target theory. If we add these two properties to the background knowledge and call SKIL again, we obtain theory T2. This is a correct definition, although more specific than the usual one.

Why does the method work? To generate the recursive clause in T2 that covers example *member(3,[4,1,3])*, SKIL needs the fact *member(3,[1,3])* corresponding to the recursive call. Although this fact does not appear among the examples, it is implied by the property *member(A,[C,A/E])* in T1.

T0: positive examples:	T1: first step theory, the properties:	T2: second step theory: (T1 is background knowledge)
member(2,[1,2,3]) *member(3,[4,1,3])*	*member(A,[C,A/E]).* *member(A,[C,E,A/G]).*	*member(A,[C,A/E]).* *member(A,[C/D]):-member(A,D).*

Table 1: Iterative bootstrap induction on the member example.

In general, the method proceeds as follows. Given positive and negative examples, we start by inducing a theory T1 invoking SKIL. If this theory performs well enough on a test set it is considered final, and it is output as the solution theory. Otherwise T1 is kept and added to the background. Then we proceed to induce T2 (invoking SKIL again). The original theory T1 is used to enhance the introduction of literals in the clauses of T2.

The performance of T2 is monitored on a test set. If it is worse than T1's, we may have reached a stable (possibly sub-optimal) point and cannot proceed. If the performance has however improved, T2 is added to the background and the process is repeated. The process stops once the given performance has been achieved.

3. Experiments

We have set out to run SKILit on a set of benchmark problems. These are simple definitions with one recursive clause and one base clause. The positive examples given need not necessarily be on the same resolution chain.

Manually selected input

Following a similar demonstration in [1], we provided SKILit with manually chosen small sets of positive and negative examples of 10 predicates, namely *append/3*,

member/2, *delete/3*, *noneiszero/1*, *plus/3*, *extractNth/3*, *factorial/2*, *rv/2*, *last_of/2*, *split/3*, and in addition also *insertion_sort/2* and *quick_sort/2*. In all of these experiments the recursive definition was successfully generated. Table 2 summarizes the results of some of these experiments.

Input: *mode(delete(+,+,-)).* *+delete(3,[1,2,3,4],[1,2,4]).* *+delete(5,[6,5],[6]).* *(-delete(3,[1,2,5],[1,2,5])).* *(-delete(7,[7,9],[7])).* Definition obtained: *delete(A,B,C):-* *dest(B,D,E),delete(A,E,F),const(C,D,F).* *delete(A,B,C):-* *dest(B,D,E), dest(E,A,F),const(C,D,F).*	*mode(extNth(+,+,-)).* *+extNth(s(0),[6,5,3,4],6).* *+extNth(s(s(0)),[1,2],2).* *(-extNth(s(s(0)),[1,2],1)).* *(-extNth(s(0),[2,1],1)).* *(-extNth(s(0),[2,1,3],1)). (-extNth(0,X,Y)).* *(-extNth(s(s(0)),[2,1,3],2)).* *(-extNth(s(s(s(0))),[1,2],2)).* *(-extNth(s(s(0)),[1],1)). (-extNth(0,X,Y)).* *extNth(A,B,C):-* *dest(B,C,D),pred(A,E),zero(E).* *extNth(A,B,C):-* *dest(B,D,E),pred(A,F),extNth(F,E,C).*
mode(rv(+,-)). *+rv([1,2,4],[4,2,1]).* *+rv([3,1],[1,3]).* *(-rv([1,2],[1])).* Definition obtained: *rv(A,B):-* *dest(A,C,D),addlast(D,C,B).* *rv(A,B):-* *dest(A,C,D),* *rv(D,E),* *addlast(E,C,B).*	*mode(qsort(+,-)).* *+qsort([3,5,2,4,1],[1,2,3,4,5]).* *+qsort([],[]). +qsort([3,1],[1,3]).* *(-qsort([2,1],[2,1])). (-qsort([1,2],[2,1])).* *(-qsort([3,1,2],[1,3,2])).* *(-qsort([3,2,1],[2,1,3])).* *qsort(A,B):-* *dest(A,C,D), part(C,D,E,F),* *qsort(E,G), qsort(F,H),* *const(I,C,H), append(G,I,B).* *qsort(A,B):- null(A), A=B.*

Table 2. Some results of SKILit with manually selected examples

Note that SKILit does not impose any constraints on the number of clauses to be generated or on the number of recursive literals, as in CRUSTACEAN. Despite this, our simple strategy permits to obtain results which are comparable to this system.

Randomly selected inputs

We also examined the ability of SKILit to synthesise theories from random sets of examples without assuming a priori knowledge of the solution.

We followed the evaluation methodology described in [1]. For each predicate we sampled positive examples from a universe of facts involving lists and peano integers. The depth of those terms varies from 0 to 4 with uniform distribution. List elements were drawn from a universe of 10 digits (0 to 9) with uniform distribution.

For each predicate, we varied the number of input positive examples from 2 to 5. The number of negative examples was kept constant. The results shown in Table 3 represent averages over 5 runs. The accuracy values were obtained by the generated definitions on relatively large independent test sets. The table also shows the percentage of output theories that contained correct recursive clauses. The last two columns give accuracies obtained by CRUSTACEAN under similar circumstances.

	Accuracy (SKILit)			*% of defs with recursion*			*CRUSTACEAN*	
predicate	*2 ex.+*	*3 ex.+*	*5 ex.+*	*2 ex.+*	*3 ex.+*	*5 ex.+*	*2 ex.+*	*3 ex.+*
append/3	*0.760*	*0.802*	*0.888*	*0*	*20*	*40*	*0.630*	*0.738*
delete/3	*0.754*	*0.880*	*1.000*	*0*	*100*	*100*	*0.617*	*0.713*
rv/2	*0.664*	*0.848*	*0.868*	*40*	*40*	*40*	*0.805*	*0.855*
member/2	*0.700*	*0.886*	*0.952*	*60*	*100*	*100*	*0.652*	*0.762*
last_of/2	*0.714*	*0.722*	*0.944*	*40*	*40*	*100*	*0.744*	*0.884*

Table 3. Some results of SKILit on randomly selected examples

We see that with random examples, SKILit got somewhat better results than the ones obtained by CRUSTACEAN. However, we believe that iterative bootstrap induction provides a more general solution which is applicable to larger classes of programs. It is potentially useful not only to learn recursion but in a more general setting of multi-predicate learning.

4. Conclusion

Iterative bootstrap induction provides a strategy for the induction of recursive theories from small sets of positive examples. The system SKILit obtained by incorporating a special case of iterative bootstrap induction within SKIL, was able to induce recursive definitions for predicates representative of an important class of logic programs.

The method presented finds regularities within the positive and expresses them in terms of the available background knowledge. Background knowledge may contain definitions of structure handling predicates, test predicates and other more complicated predicates. This fact allows SKILit to express the regularities in a richer language than in [1], enabling it to handle larger classes of problems.

References

[1] Aha D W, Lapointe S, Ling C X, Matwin S (1994): "Inverting Implication with Small Training Sets", in *Proceedings of the European Conference on Machine Learning, ECML-94*, ed. F. Bergadano and L. de Raedt, Springer Verlag.

[2] Brazdil P, Jorge A. (1994): "Learning by Refining Algorithm Sketches", in *Proceedings of ECAI-94*, T. Cohn (ed.), Amsterdam, The Netherlands.

[3] Cohen W W (1993): "Rapid prototyping of ILP systems using explicit bias" in *Proceedings of 1993 IJCAI Workshop on ILP.*

[4] Idestam-Almquist P (1993) "Generalization under implication by recursive anti-unification", in *Proceedings of ILP-93*, Jozef Stefan Institute, technical report.

[5] Michalski R S, (1994): "Inferential Theory of Learning: Developing Foundations for Multistrategy Learning", in *Machine Learning, A Multistrategy Approach, Volume IV*, ed.by Ryszard Michalski and Gheorghe Tecuci, Morgan Kaufmann.

[6] Muggleton S. (1993): "Inductive Logic Programming: derivations, successes and shortcomings" in *Proceedings of ECML-93*, P.Brazdil (ed.), Springer-Verlag.

[7] Zelle J M, Mooney R J, Konvisser J B, (1994):"Combining Top-down and Bottom-up Techniques in Inductive Logic Programming" in *Proceedings of the Eleventh International Conference on Machine Learning ML-94*, Morgan-Kaufmann.

[8] Richards B, Mooney R (1992): "Learning relations by pathfinding" in *Proceedings of the Tenth National Conference on Artificial Intelligence*, Cambridge, MA, MIT Press.