4 J − 1

Examples' Depth and Induction of Recursive Logic Programs

Edson Satoshi GOMI　Mitsuru ISHIZUKA
東京大学 工学部 電子情報工学科
e-mail: gomi , ishizuka @miv.t.u-tokyo.ac.jp

1 Introduction

The main goal of a Inductive Logic Programming (ILP) system is to induce a logic program that explains a given set of positive examples and is consistent with negatives examples. Inductive Logic Programming inherist concepts and methods from Machine Learning and Logic Programming.

An important class of logic programs is the class of recursive programs. Many programs for list processing and mathematical functions are expressed using recursive logic programs. Recursive programs consist of at least two clauses, which are named base and recursive clauses. A recursive clause has one or more literals in its body with the same predicate of clause's head.

It has not been possible to induce correct recursive programs, using existing ILP systems, without carefully selected set of training examples. An analysis of these training sets shows that the depth of an examples plays an important role on induction of recursive programs: if examples of any depth k aren't present in the given set of examples, induction can't be performed correctly. The main reason for this problem is that existing ILP systems use induction operators based on ***θ-subsumption*** instead of operators based on ***implication***. Operators based on θ-subsumption are incomplete with respect to implication.

In order to overcome this problem, we use an approach called *forced simulation*. Basically, the forced simulation algorithm simulates (in a controlled way) the hypothesized recursive program for each example. This is a simple technique to *invert implication*

2 The model-theoretic approach

Given background knowledge B and a set of positive (E^+) and negative (E^-) examples, the goal of an ILP system is to find an hypothesis H such that the following four equations hold:

$$
\begin{array}{rll}
B \not\models E^+ & \text{Prior Necessity} & (1)\\
B \wedge H \models E^+ & \text{Posterior Sufficiency} & (2)\\
B \wedge E^- \not\models \square & \text{Prior Satisfiability} & (3)\\
B \wedge H \wedge E^- \not\models \square & \text{Posterior Satisfiability} & (4)
\end{array}
$$

Dept. of Information and Communication Engineering,
Faculty of Engineering, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113, JAPAN

Applying the deduction theorem to equation 2, the following relation can be obtained:

$$B \wedge \overline{E^+} \models \overline{H} \qquad (5)$$

Equation (5) provides a method to generate hypothesis using background knowledge and positive examples. Since H and E^+ are clauses, $\overline{H}$ and $\overline{E^+}$ are conjunctions of ground skolemised literals. Using a sound derivation method to obtain all ground literals entailed by $B \wedge \overline{E^+}$, it is possible to deduce the most specific solution for hypothesis H.

To illustrate this approach, let's consider the list membership problem. List membership program is a example of a recursive program, and suppose the base clause is already known and it is incorporated in the background knowledge ($B = \{member(X, [X \mid Y])\}$). Let's consider the example $E^+ = member(b, [a, b, c])$. The successive application of equation 5 gives:

$$
\begin{array}{rcl}
B \wedge \overline{E^+} & \models & \overline{member(b,[a,b,c])}\\
 & \models & member(a,[a,b,c])\\
 & \models & member(b,[b,c])\\
 & \models & member(c,[c])\\
 & \models & \cdots
\end{array}
$$

The most specific solution is:

$$
\begin{array}{rcl}
member(b,[a,b,c]) & \leftarrow & member(a,[a,b,c]),\\
 & & member(b,[b,c]),\\
 & & member(c,[c]),\ldots \quad (6)
\end{array}
$$

The most specific hypothesis is denoted as $\perp$ and can be infinite, as shown in solution (6). To construct a finite most specific hypothesis, it is necessary to limit the number of resolutions allowed to generate solutions.

The induction process is transformed in a search problem through the set of clauses more general than the most specific hypothesis. This search can ben executed using a top-down approach, starting from the most general solution ($\square$), or using a bottom-up approach, starting from the most specific hypothesis.

3 Example's depth

The depth of an example is an important concept and is defined as follows:

Definition 1 *An example has* depth *0 if it is a base case, otherwise it is a recursive case and has depth k if it is located at k resolution steps from the base case.*

Consider the following deduction chain of recursive program *append*:

- Program *append*:

$$\begin{aligned}&append([\,], List, List) \leftarrow\\&append([H|Tail1], List2, [H|Tail3]) \leftarrow\\&\quad append(Tail1, List2, Tail3)\end{aligned}$$

- Ground literals generated during the proof process of $append([1,2,3,4],[5],[1,2,3,4,5])$ and their depth values:

Depth 4	$append([1,2,3,4],[5],[1,2,3,4,5])$
Depth 3	$append([2,3,4],[5],[2,3,4,5])$
Depth 2	$append([3,4],[5],[3,4,5])$
Depth 1	$append([4],[5],[4,5])$
Depth 0	$append([\,],[5],[5])$

Usually ILP systems perform generalization under θ-subsumption instead of generalization under implication. The main reason is that it is more easy to compute generalization under θ-subsumption. However, θ-subsumption doesn't substitute completely implication. If clause C θ-subsumes clause D ($C \succeq D$) then $C \models D$, but the opposite doesn't hold for recursive cases. Therefore, θ-subsumption can be used only for one resolution step. For this reason, when using induction methods based on θ-subsumption it is necessary to give a complete set of positive examples representing examples from depth 0 to depth n. Recursive logic programs won't be induced correctly if there is lack of any example of depth k ($0 \leq k \leq n$).

4 Induction of Recursive Programs

To solve the problem of induction of recursive logic programs we use an approach named forced simulation [2)]. A restricted class of recursive programs is considered: the class of *two-clause closed linear recursive ij-determinate* programs. Although this class is quite restricted, it includes many usefull logic programs, like list membership, append, reverse, prefix, suffix, etc. Basically, the forced simulation algorithm simulates the computation of an hypothesis, in order to verify if it is correct.

The base and recursive clauses of the initial hypothesis are constructed using the base examples (examples of depth 0) and recursive examples (the remaining examples). To separate the set of positive examples in base and recursive cases, we use the method of *minimal multiple generalization* [1)] (*mmg*). The *mmg* algorithm can be considered as an extension of the *least general generalization* (*lgg*). For example, given a set of positive examples, the *lgg* algorithm finds precisely one literal that covers the given set of examples. On the other hand, the *mmg* algorithm finds two literals, $(\{append([\,], X, X),\ append([X|Y], Z, [X|W])\})$ each covering a different group of examples (the base and recursive cases).

In the next step, the operation of ***relative least general generalization*** [3)], with respect to background knowledge, is applied over the set of base and recursive examples, and the operation of ***flattening*** [4)] is applied to eliminate all function symbols. The recursive literal to be inserted in the recursive clause is constructed using variables of the flattened recursive clause. At this point, the forced simulation algorithm starts to simulate the constructed program for all positive examples. For each example of depth n, the forced simulation algorithm generates examples of depth k ($0 \leq k \leq n$). If the constructed hypothesis is correct, the forced simulation algorithm will be able to prove all positive examples. If the program is not correct, another recursive literal is constructed and the process is repeated again.

5 Concluding remarks

This paper presented the problems concerned with induction of recursive logic programs, when using induction operators based on θ-subsumption, which is incomplete with respect to implication. A consequence of this incompleteness is that sets of examples need to be carefully constructed, without lack of examples of depth k ($0 \leq k \leq n$). To overcome this problem, we use an approach called forced simulation. This algorithm can generate examples of depth k, even if they aren't present in the initial set of examples.

References

1) H. Arimura, T. Shinohara, and S. Otsuki. Polynomial time inference of unions of tree pattern languages. In *Proceedings of the Workshop on Algorithmic Learning Theory*, p. 105–114, 1991.

2) William W. Cohen. Pac-learning a restricted class of recursive logic programs. In *AAAI-93*, pages 86–92, 1993.

3) S. Muggleton. Efficient Induction of Logic Programs. In *Inductive Logic Programming*, pages 281–298, Academic Press, 1992.

4) Céline Rouveirol. Flattening and saturation: Two representation changes for generalization. *Machine Learning*, 14:219–232, 1994.