A new algorithm to automate inductive learning of default theories

Xinyu Zhang, Yuyao Wang

*Stony Brook University, New York, USA*

**Abstract**

An algorithm in inductive learning of broad concepts should be able to distinguish concept examples from noisy data and exception. One approach for finding patterns in exceptions is correspond to the problem of learning default theories. For our project, we re-implemented an algorithm to this problem introduced by a paper in TPLP.

**KEYWORDS:** inductive learning, default theories.

**Introduction**

Given the background knowledge (KB) and a set of positive and negative examples, Inductive Logic Programming (ILP) should learn theories in the form of Horn logic programs. However, ILP have some drawbacks, first, Horn clause are not sufficiently expressive for representing and reasoning when the background knowledge is incomplete. In addition, ILP is not able to handle exception to general rules because they can only learn rules under the assumption that there are no exceptions in the training data (KB), in other words, ILP can not distinguish exceptions and noisy data, so those data will be treated in the same manner. But, in fact, we can also learn something from those data because they might also follow some certain kind of pattern or rule.

To solve this problem, the paper introduced FOLD algorithm. This algorithm is an extension of FOIL algorithm and support both categorical and numeric features. Also, this algorithm can learn recursive rules, if needed, FOLG will introduce new predicates.

**Problem Description**

**Given:**

* Background theory B, in the form of a normal logic program, clause of the form h <- l1, l2, …, lm, and not lm+1, not lm+2, …, not ln. h and l1, l2, …, lm are positive literals and not denotes NAF with stable model semantics.
* Two disjoint sets of grounded goal predicates E+ and E-, these are positive and negative examples respectively.
* A hypothesis language of predicates including function and atom free predicates. It also contains a set of arithmetic constraints of the form {A<= h, A>= h}, A is a variable and h is a real number.
* A convers (H, E, B) function, which returns the subset of E that is extensionally implied by the current hypothesis H, given the background knowledge B.
* A score (E+, E-, H, B), which specifies the quality of the hypothesis H with respect to E+, E-, B.

**Find:**

* A theory for which convers (H, E+, B) = E+ and convers (H, E-, B) = []

**Example:**

***B: E+:***

bird(X) ← penguin(X). fly(a). fly(b). fly(e). fly(f). fly(g). fly(h).   
penguin(X) ← superpenguin(X).

bird(a). bird(b).

penguin(c). penguin(d).  
superpenguin(e). superpenguin(f). cat(c1).

plane(g). plane(h). plane(k). plane(m).

damaged(k). damaged(m).

**FOLD algorithm learns the following theory:**

fly(X) ← plane(X), not ab0(X).

fly(X) ← bird(X), not ab1(X).

fly(X) ← superpenguin(X).

ab0(X) ← damaged(X).

ab1(X) ← penguin(X).

**Information Gain**

The FOLD loop of the program will try to find a clause with the highest IG value using a general-to-specific hill-climbing search. The most general clause is p(,…, ) 🡨 True, predicate p/n is the predicate being learned and each is a variable. The refinement operator specializes the current clause h ← b1, . . . bn. This is realized by adding a new literal l to the clause yielding h ← b1, . . . bn, l.

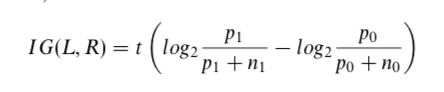


Figure1 IG

L is the candidate literal to add to rule R, is the number of positive examples implied by the rule R, n0 is the number of negative examples implied by the rule R, p1 is the number of examples implied by the rule R + L, n1 is the number of negative examples implied by R also covered by R + L.

**FOLD algorithm**

The target of FOLD algorithm is to learn a concept from given information as a default theory and possibly some exceptions. The major steps of FOLD algorithm are as follow, for the first step, FOLD will specialize a general rule with positive literals:

Goal (, ..., ) ←True

No negative literal is used at this stage. Once the IG value become 0, this process stop. Until this process over, if there are still some negative exampled covered, they must be noisy data of exception to current learned rule.

We might obtain nothing from noisy data, but we can learn something from negative examples, to achieve this goal, FOLD swaps the positive and negative examples recursively calls the FOLD algorithm to learn the exception rules(s). Each time a rule is discovered for exception, a new, unique predicate (avoid naming collision) will be introduced:

ab (, ..., )

In the case of noisy data or the presence of uncertainty due to the lack of information, FOLD will enumerate for the purpose of training converge and detect noisy data sample.

This figure shows the core steps of FOLD algorithm:

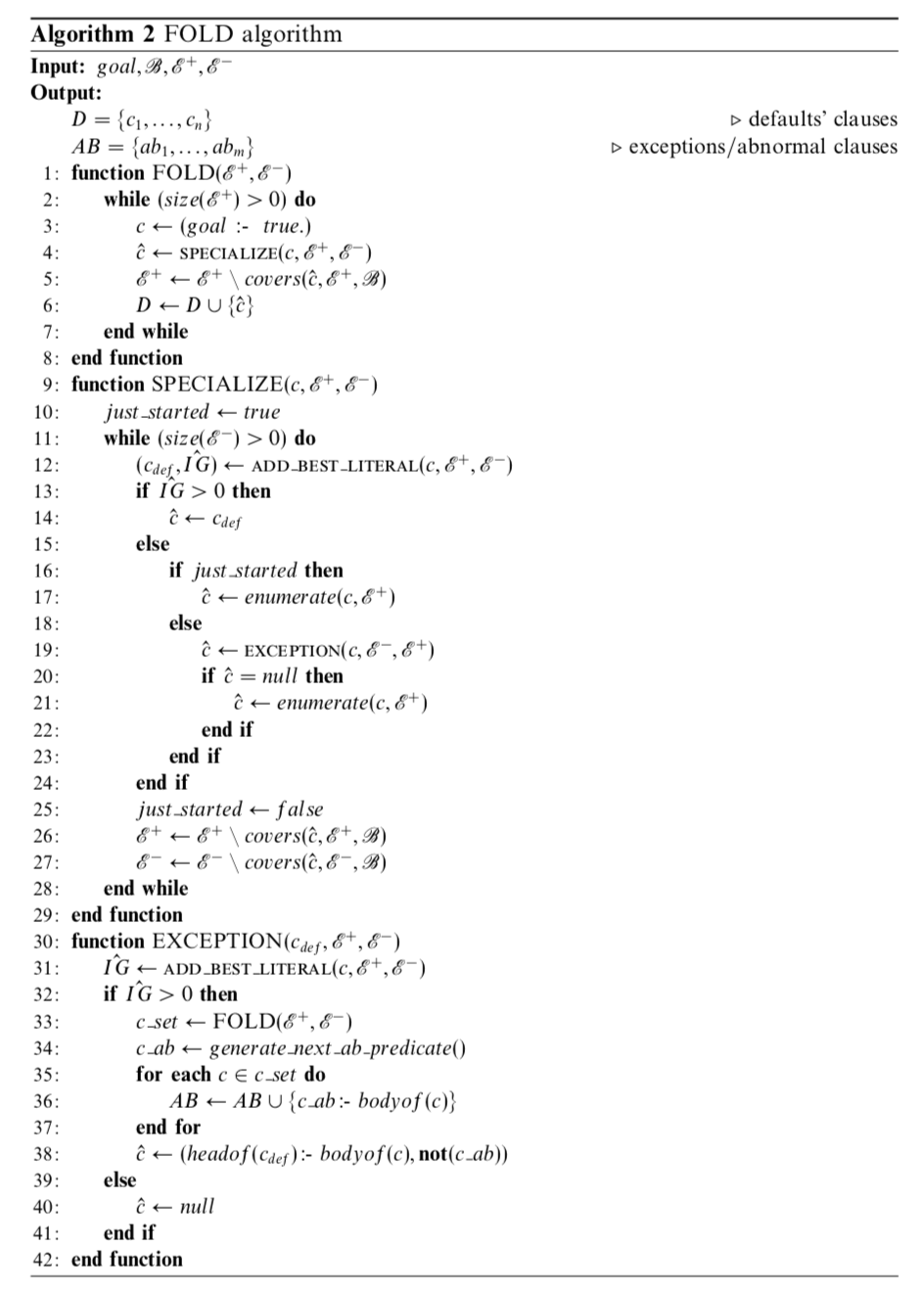


Figure2 FOLD algorithm

FOLD starts with the most general clause (e.g. fly(X) ← true) In line 4 of Figure 2, this clause is refined by calling the function specialize/5.

In line 5, 6 of Figure 2, set of positive examples and set of discovered clauses are updated to reflect the newly discovered clause.

**Function**: SPECIALIZE (*c*, ***E+***, ***E-***)

compute the *IG* (i.e. *IG =* *score (****E+****,* ***E−****,* ***H****,* ***B****)*) of each literal

if *IG* > 0

return this literal (named as *cdef*)

else call *c’ =* EXCEPTION (*cdef*, ***E-***, ***E+***)

update ***E+*** and ***E-*** with *c’* and***B***respectively

**Input**: general goal *c*, ***B***, ***E+***, ***E-***

**Output**:

the literal *c’* with highest *IG* value

In line 12, the program will assign a literal with greatest IG value. In line 13-24, depending on the value of IG, either the positive literal is accepted or call the exception/4. If the IG for the first iteration becomes 0, then a clause that just enumerate all the positive examples is produced. The paper used a flag called “just\_started” to represent the first iteration, we also used this method.

For line 26, 27, the set of positive and negative examples are updated, this is to reflect the changes of the current clause.

**Function**:EXCEPTION (*c*, ***E+***, ***E-***):

compute the *IG’* of each literal w.r.t. *cdef*. Select the one with greatest *IG’* value.

if *IG’ > 0*

recursively call FOLD (*cbest*,***E+***, ***E-***), get Dset

generate ab\_predicate cab

**Input**: *cdef*, ***B***, ***E+***, ***E-***

**Output**:

*AB = {ab1 ,… , abm}* //exceptions/abnormal clauses

headof(cdef) :- bodyof(cdef), not(cab)

EXCEPTION function uses original negative examples as its positive examples because it wants to find a rule for exceptions, FOLD is recursively called to extract this rule.

After recursively calling FLOD to extract the rule(s), new ab predicate will be produced and it will be combined with the body of the rule(s) found by recurring FOLD. In line 38 of Figure 2, default and exception will be attached together to form a single clause.

**Implementation**

**Input:**

**Output:**

**Conclusion:**

*GitHub link:*