CSE505 - Computing with Logic

A New Algorithm to Automate Inductive Learning of Default Theories

FARHAD SHAKERIN, ELMER SALAZAR, GOPAL GUPTA
ICLP, August 2017

Wei-Cheng Li
112073486

December 17, 2018

1 Introduction

In an inductive learning programming (ILP) process, the program learns theories from the positive and negative examples and discovers the corresponding rules. However, the typical ILP systems treat the exceptions and the noise in the same way. ILP cannot handle the exceptions whereas they sometimes follow a pattern which can also be learned to a general rule. For example, we want to learn the concept of birds' flying ability. We know that in most cases birds can fly, but there are some birds such as penguins and ostriches that are the exceptions. Here, ILP systems will either fail to determine a general rule or cover the positive examples without caring much about negative ones. Other algorithms, such as First Order Inductive Learning (FOIL), induces rules that are non-constructive, thus it is sometimes not intuitive at all.

To tackle this problem, the authors introduced new algorithms, which are the extension of FOIL, called First Order Learner of Default (FOLD)[1] and FOLD-R to handle categorical and numeric features respectively. Both of them learn non-monotonic stratified logic program which allows negation-as-failure (NAF).

2 Background

2.1 Problem Statement

We follow the statements provided in the paper. For an inductive non-monotonic logic programming problem, it can be formalized as below:

Given

- a background theory B in the form of a normal logic program, i.e. clauses of the form $h \leftarrow l_1, ...l_m, \neg l_{m+1}, ..., \neg l_n$, where $l_1, ..., l_n$ are positive literals.

- two disjoint seets of grounded goal predicates $E+$ and $E-$ as positive examples and negative examples respectively.
- a function $covers(H, E, B)$ that returns the subset of E which is implied by the hypothesis H given the background knowledge B.
- a function $score(E+, E-, H, B)$ that specifies the quality of the hypothesis H with respect to $E+$, $E-$, B.

Objective

- find a theory T that makes $covers(T, E+, B) = E+$ and $covers(T, E-, B) = \phi$

Namely, we want to find a set of rules, which satisfies all of the positive examples and rules out all of the negative examples.

2.2 FOIL Algorithm

The FOLD algorithm is an extension of the FOIL algorithm (Quinlan 1990). The FOIL algorithm is a top-down ILP system following a sequential covering approach to induce a hypothesis. The algorithm can be summarized as Algorithm 1. Basically, FOIL repeatedly searches for the clauses with the greatest score with respect to $E+$, $E-$, and the hypothesis H at that time.

Algorithm 1 FOIL Algorithm Summary

Input: goal, B, $E+$, $E-$
Output: theory T

```
1: function FOIL(goal, B, E+, E−)
2:     initialize T ← φ
3:     while size(E+) > 0 do
4:         c ← (goal :- true.)
5:         while size(E−) > 0 do
6:             for each candidate literal l do
7:                 compute score(E+, E−, H ∪ {c'}, B)
8:             conjoin l with the best score to c.body
9:             E− ← E−\covers(¬c, E−, B)
10:        add c to T
11:        E+ ← E+\covers(c, E+, B)
12:    return T
```

The score in line 7 is the information gain (IG) of the corresponding rule, which is calculated as follows (Mitchell 1997):

$$IG(L, R) = t(log_2 \frac{p_1}{p_1 + n_1} - log_2 \frac{p_0}{p_0 + n_0})$$

where L is the candidate literal to conjoin to the body of the rule R, p_0 is the number of positive examples implied by R, n_0 is the number of negative examples implied by R, p_1 is the number of positive examples implied by $L + R$, n_1 is the number of negative examples implied by $L + R$, and t is the number of positive examples implied by R also covered by $R + L$.

Basically, the clause with the "best" literal added increases the largest number of positive examples that satisfy the rule. Nevertheless, this approach cannot capture the exceptions and the result somtimes is not much intuitive for humans.

3 Method

Like the FOIL algorithm, the FOLD algorithm repeatedly explores new literal and conjoins it to the current rule theory. The difference is that, first, FOLD do not consider negative literals when exploring. Second, when there is no improvement with the remain literals, FOLD then tries to find the rules governing negative examples. This can be viewed as a subproblem by swappping the postive and negative examples, so we just call the FOLD algorithm recursively. The below steps shows the progress of how it works:

1. The function $FOLD(goal, E+, E-, B)$ returns a set of rules that satisfies all $E+$ and rules out all $E-$, with the given background knowledges. The rule starts with the most general one, i.e., *goal* :- *true*.

2. For each iteration, FOLD picks the literal with maximum information gain. That is, we maintains the coverage as many positive examples as possible.

3. Step 2 stops when the information gain is zero.

4. If $|E-|$ is not zero, they are either noisy data or exceptions. Then, we try to find the rules governing the negative examples. This step is called *exception* and can be done by calling $FOLD(goal, E-, E+, B)$ (step 1).

5. In *exception* step, when we find a rule for the exceptions, we create a new predicate and add the negation of it to the rule set.

6. If no pattern can be learned due to the lack of information or the noisy data, FOLD then enumerates the rest positive examples.

4 Project Work

I implemented the FOLD algorithm with several functions in SWI-Prolog, and then test them with several different types of cases.

4.1 Functions

1. **fold (Goal, Pos, Neg, Background, Predicates, D, AB, IsExcept)**
   The main function of the algorithm. We start with **Goal** = *goal* :- *true* and finally return the set of rules **D** and **AB**. **D** denotes the default clauses while **AB** denotes the abnormal clauses (exceptions). **IsExcept** checks whether the current process is called in the exception step.

2. **fold_loop (Goal, Pos, Neg, IsExcept, Prev, D)**
   This is the looping function which repeatedly calls the specialize function. It stops when **Pos** is empty. Every time the specialize function returns a new clause, we append it to **Prev**. Eventually, we assign **D** to **Prev** when the loop stops.

3. **specialize (Pos, Neg, Clause0, PosClause, Just_started)**
   Find and return a rule as **PosClause** based on positive examples (**Pos**), negative examples (**Neg**), hypothesis (**Clause0**), and the background knowledge. **Just_started** checks if it is the first call to determine the next action (enumerate or exception) if we get IG=0.

4. **exception ((Goal :- Body), Pos, Neg, Clause1)**
   Call fold function to get the exception rules governing **Neg**. After finding the rules, we create a new predicate and append it to **Body**, becoming **Clause1**. The predicate is also added to the abnormal clauses set.

5. **enumerate (Clause0, Pos, Clause1)**
   The function enumerates the positive clauses. We create a clause $member/2$ which is also the built-in function in Prolog to represent the enumerating process. We then add it to the body of **Clause0** as **Clause1**.

6. **compute_gain (Neg, Pos, Info, Clause, Gain)**
   Return the IG value given **Clause**, the examples **Pos** and **Neg**, and the gain value **Info** representing IG without **Clause** added to the rule.

4.2 Tools

There are some other functions to help the main functions manage the important tasks.

1. **unify_arg(L, Head, Result)**
   To create a predicate, we have to make the variables consistent in both the head and the body. This function unifies the variable in literal **L** and the one in the predicate's head **Head**.

2. **add_best_literal(Clause0, Pos, Neg, Clause1, IG)**
   This function is used in the specialze function when we have to pick a best literal in every turn. Basically it calculate every litaral candidate with the compute_gain/5 function, and return the greatest one with its **IG**.

3. **choose_tie_clause(Clause, B0, B1, C)**
   If we find there are multiple literal candidates with the same information gain value, we choose the one with the least constraints. We append the literals **B0** and **B1** to **Clause** respectively and pick up the one with less number of variables.

4. **covered_examples((A:-B), Xs, Xs1)**
   This function acts as $covers(H, E, B)$. Given the predicate **(A:-B)** and the example **Xs**, we filter out the unsatisfied examples and return the rest as **Xs1**.

5. **uncovered_examples((A:-B), Xs, Xs1)**
   Similar with the above, but this time we filter out the satisfied elements and return the rest as **Xs1**.

5 Testing

To verify the algorithm, I used the testing data provided in the paper, each of which is designed under different conditions, including the noisy data and the exceptions. Moreover, I further used a more complicated dataset outside the paper to see if the algorithm can still achieve a high accuracy.

5.1 Single Exception

Input:

Background: bird(X) ← penguin(X).
bird(tweety). bird(et).
cat(kitty). penguin(polly).
Example+: {tweety, et}
Example-: {kitty, polly}
Goal: fly(X)

Output:

D: fly(X) ← bird(X), \+ab0(X).
AB: ab0(X) ← penguin(X).

5.2 Enumerate

Input:

Background: bird(X) ← penguin(X).
bird(tweety). bird(et).
cat(kitty). penguin(polly).
Example+: {tweety, et, **jet**}
Example-: {kitty, polly}
Goal: fly(X)

Output:

D: fly(X) ← bird(X), \+ab0(X).
fly(X) ← member(X, [jet]).
AB: ab0(X) ← penguin(X).

5.3 Multiple and Nested Exceptions

Input:

Background: bird(X) ← penguin(X).
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).
Example+: {a, b, e, f, g, h}
Example-: {c, d, c1, k, m}
Goal: fly(X)

Output:

D: fly(X) ← plane(X), \+ab1(X).
fly(X) ← superpenguin(X).
fly(X) ← bird(X), \+ab0(X).
AB: ab0(X) ← penguin(X).
ab1(X) ← damaged(X).

5.4 Mistake Results

Given the dataset provided by Cornell University[2], the output does not actually cover all of the examples. In fact, there are 2 mistakes among 14 examples. Moreoever, the generated answer does not seem to give intuitive rules.

Input:

Day	*Outlook*	*Temperature*	*Humidity*	*Wind*	*Ski?*
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Figure 1: A skiing example from [2]

Output:

D: ski(X) ← member(X, [d4]).
ski(X) ← high(X), \+ab10(X).
ski(X) ← mild(X), \+ab1(X).
ski(X) ← normal(X),\+ab0(X).
ski(X) ← overcast(X).

AB: ab0(X) ← strong(X).
ab1(X) ← rain(X).
ab1(X) ← weak(X).
ab10(X) ← high(X).
ab10(X) ← sunny(X).

6 Modification

Based on the steps of algorithms in the paper, I somehow modified some parts of the algorithm for the original one did not work when running the testing data. Perhaps the authors made some typos in the paper.

First, in my code, the specialize function does not stop at IG=0. Instead, I changed the condition to -1. The reason is that if $n_0 = 0$ and $p_1 > 0$ after we found a rule $L + R$, the value of IG would be $log_2 1 - log_2 1 = 0$, but actually we do not expect to go to exception function in this round for we had found a rule governing some positive examples. To fix this problem, only until there is no rule covering a subset of $E+$ do we turn to exception function, and we use -1 as the signal of it.

Second, in line 27 of the FOLD psuedo code, the algorithm updates $E-$ at the end of every iteration in the specialize function. It eliminates the examples covered by the given clause. However, the uncovered examples should be removed, not the covered ones. Only leaving the covered examples can help us know if there exists an exception. Thus, I designed two functions, *covered_examples*/3 and *uncovered_examples*/3, to handle two different requests.

7 Conclusion

In the previous first three tests, we can see that the algorithm successfully learned the theories of the dataset without the influnce of the noisy data or the exceptions. However, when the background knowledge becomes complicated like the skiing example, the algorithm caused some mistakes (2 out of 14) when we try to verify the knowledge with the generated rules. Besides, there are still some drawbacks in my implementation, such as the clauses high(X) and \+high(X) exist at the same body, which is a contradiction and will always return false.

8 GitHub Link

https://github.com/zon5566/CSE505-Final-Project

References

[1] Farhad Shakerin, Elmer Salazar, and Gopal Gupta, *A New Algorithm to Automate Inductive Learning of Default Theories*, International Conference on Logic Programming (ICLP), Melbourne, 2017.

[2] Thorsten Joachims, CS478 - Machine Learning, Cornell University, Spring 2008. *<http://www.cs.cornell.edu/courses/cs478/2000SP/lectures/rule-learning.pdf>*