CSE 505 - Computing with Logic
Project - Phase 3

Charuta Pethe
111424850

December 20, 2017

1 Introduction

The paper that I have chosen is *A New Algorithm to Automate Inductive Learning of Default Theories* [1], by Farhad Shakerin, Elmer Salazar and Gopal Gupta from The University of Texas at Dallas, Texas, USA. This paper was published in the 33rd International Conference on Logic Programming (ICLP) at Melbourne, Australia in August 2017.

2 Paper Methodology

2.1 Background

Classical machine learning methods involve predictive models that are algebraic solutions to optimization problems. These methods are neither intuitive nor easily understandable, and are hence hard to justify when applied to a new data sample. Moreover, upon addition of new knowledge, the entire model needs to be relearned.

However, Inductive Logic Programming (ILP) [2] is a technique where the model is in the form of Horn clauses that are more comprehensible, and also allow the knowledge base to be incrementally extended without having to relearn the entire model. However, as negation-as-failure cannot be represented using Horn clauses, ILP is insufficient for reasoning in the absence of complete background knowledge. In addition, ILP cannot handle exceptions to rules, which results in exceptions being treated the same as noise.

The exceptions to rules often follow a pattern themselves, and can be learned. The resulting theory that is learned is a default theory, which describes the underlying model more accurately, and is more intuitive and comprehensible.

The paper presents two algorithms for learning default theories:

1. FOLD (First Order Learner of Default) - To handle categorical features
2. FOLD-R - To handle numeric features.

2.2 Problem Statement

The problem that is tackled in the paper can be formalized as follows:

Given

- a background theory $\boldsymbol{B}$ in the form of a normal logic program, i.e clauses of the form $h \leftarrow l1, \ldots, lm, not\ lm{+}1, \ldots, not\ ln$, where h and $l1, \ldots, ln$ are positive literals
- two disjoint sets of grounded goal predicates $\boldsymbol{E+}$ and $\boldsymbol{E-}$ (positive and negative examples respectively)
- a hypothesis language of predicates $\boldsymbol{L}$
- a function ***covers****(H, E, B)* which returns the subset of E that is extensionally implied by the current hypothesis H, given the background knowledge B

Find

- a theory $\boldsymbol{T}$, for which *covers(T, E+, B)* = *E+* and *covers(T, E-, B)* = ϕ

2.3 FOLD Algorithm

The FOLD algorithm learns a concept as a default theory, along with exceptions. A brief description of the steps in the algorithm is as follows:

1. FOLD first tries to learn the default theory by specializing a general rule of the form $goal(V_1, ..., V_n) \leftarrow true$.
2. Each specialization rules out some already covered negative examples without decreasing the number of positive examples covered significantly.
3. This process stops once the information gain (as explained in Section 2.4) becomes zero.
4. If some negative examples are still covered at this point, they are either noisy data samples or exceptions to the rule learned so far. Finding the set of rules governing the set of negative examples is now a subproblem.
5. To solve this subproblem, FOLD swaps the current positive and negative examples, and recursively calls itself to learn the exception rules.
6. Each time a rule is discovered for a set of exceptions, a new predicate is introduced, and the negation of the predicate is added to the original rule.
7. If there is noisy data or uncertainty due to lack of information, there is no pattern to learn. In this case, FOLD enumerates the positive examples.

The paper proposes the following proofs about the algorithm:

1. **Finiteness:** The FOLD algorithm terminates on any finite set of examples.
2. **Soundness:** The FOLD algorithm always learns a hypothesis that covers no negative examples.
3. **Completeness:** The FOLD algorithm always learns a hypothesis that covers all positive examples.

2.3.1 FOLD: Example 1

Input

B: $bird(X) \leftarrow penguin(X).\ bird(tweety).\ bird(coco).\ cat(kitty).\ penguin(polly).$
Goal: To learn the rule $fly(X)$.
E+: $\{tweety,\ coco\}$
E-: $\{kitty,\ polly\}$

Output

$fly(X) \leftarrow bird(X),\ not\ ab0(X).$
$ab0 \leftarrow penguin(X).$

2.3.2 FOLD: Example 2

FOLD can also learn the correct theory in presence of nested exceptions.

Input

B: $bird(X) \leftarrow penguin(X).\ penguin(X) \leftarrow superpenguin(X).$
$bird(a).\ bird(b).\ penguin(c).\ penguin(d).\ superpenguin(e).\ superpenguin(f).$
$cat(c1).\ plane(g).\ plane(h).\ plane(k).\ plane(m).\ damaged(k).\ damaged(m).$
Goal: To learn the rule $fly(X)$.
E+: $\{a,\ b,\ e,\ f,\ g,\ h\}$
E-: $\{c,\ d,\ c1,\ k,\ m\}$

Output

$fly(X) \leftarrow plane(X),\ not\ ab0(X).$
$fly(X) \leftarrow penguin(X),\ not\ ab1(X).$
$fly(X) \leftarrow superpenguin(X).$
$ab0 \leftarrow damaged(X).$
$ab1 \leftarrow penguin(X).$

2.4 Information Gain

As explained in Section 2.3, the literal which has the maximum information gain is chosen and added to the set of rules. It is calculated as follows:

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

Here:
L is the candidate literal to add to rule R
p_0 is the number of positive examples implied by the rule R
n_0 is the number of negative examples implied by the rule R
p_1 is the number of positive examples implied by the rule $R + L$
n_1 is the number of negative examples implied by the rule $R + L$
t is the number of positive examples implied by the rule R and covered by the rule $R + L$

3 Project Work

I have implemented, tested and verified the FOLD algorithm. My work and observations are described in the subsequent sections. The code for my implementation of the algorithm is available on:

3.1 Algorithm Implementation

I have implemented the FOLD algorithm in Python 2.7. I have implemented the following functions:

- **Described in the paper:**
  1. ***FOLD***
     - **Input:** positive examples, negative examples, predicates, background knowledge
     - **Output:** default rules, abnormal rules
     - **Description:** This function learns and returns the rules governing a goal, using the inputs.
  2. ***specialize***
     - **Input:** rule, positive examples, negative examples, predicates, background knowledge
     - **Output:** specialized rule
     - **Description:** This function takes a rule, refines and returns it.
  3. ***exception***
     - **Input:** rule, positive examples, negative examples, predicates, background knowledge
     - **Output:** rule with learned exception
     - **Description:** This function recursively calls the FOLD algorithm to learn an exception to a rule.
- **Implemented on my own:**
  1. ***addBestLiteral***
     - **Input:** rule, positive examples, negative examples, predicates, background knowledge
     - **Output:** new rule, its associated information gain
     - **Description:** This function selects the best clause to add to the body of the rule, i.e. the clause which gives highest information gain, and returns it.
  2. ***covers***
     - **Input:** rule, examples, background knowledge
     - **Output:** new examples

- **Description:** This function returns a subset of the given examples covered by the given rule, given the background knowledge.

3. ***coversHelper***
   - **Input:** rule, examples, background knowledge
   - **Output:** True / False
   - **Description:** This function returns whether a rule covers an example, given the background knowledge.
4. ***enumerateFunction***
   - **Input:** rule, examples
   - **Output:** new rule
   - **Description:** This function enumerates all examples, adds the list to the body of the rule, and returns it.
5. ***generate_next_ab_predicate***
   - **Input:** -
   - **Output:** string (predicate name)
   - **Description:** This function generates the name of the next abnormal predicate and returns it.

3.2 Algorithm Testing

I have tested the FOLD algorithm on the examples described in the paper as well as my own examples. These examples include:

1. Learning rules where there is a single exception
2. Learning rules where there are multiple exceptions to the same rule
3. Learning rules which have nested exceptions
4. Learning rules where examples have to be enumerated if no underlying pattern is found.

3.3 Results

The test examples along with expected and observed outputs, and their comparisons, have been described in the subsequent sections.

3.3.1 Example 1 - Exception

- **Input:** All penguins are birds. Tweety and Et are birds. Kitty is a cat. Polly is a penguin. Tweety and Et can fly. Kitty and Polly cannot fly.
- **Expected output:** All birds can fly, except penguins.
- **Observed output:** It is the same as the expected output.

3.3.2 Example 2 - Enumeration

- **Input:** All penguins are birds. Tweety and Et are birds. Kitty is a cat. Polly is a penguin. Tweety, Et and Jet can fly. Kitty and Polly cannot fly.
- **Expected output:** All birds can fly, except penguins. Jet can fly.
- **Observed output:** It is the same as the expected output.

3.3.3 Example 3 - Nested exception

- **Input:** All penguins are birds. All superpenguins are penguins. A and B are birds. C and D are penguins. E and F are superpenguins. C1 is a cat. G, H, K and M are planes. K and M are damaged. A, B, E, F, G and H can fly. C, D, C1, K, M cannot fly.
- **Expected output:** All birds can fly, except penguins. However, superpenguins can fly. All planes can fly, except for those that are damaged.
- **Observed output:** It is the same as the expected output.

3.3.4 Example 4 - Blood donation compatibility

- **Input:** p1-p4 are O+. p5-p8 are O-. p9-p12 are A+. p13-p16 are A-. p17-p20 are B+. p21-p24 are B-. p25-p28 are AB+. p29-p32 are AB-. A set of who can and cannot donate for each blood group is also provided.
- **Output:** The expected output is a set of rules for each predicate. Here, the following outputs are expected and observed:
  - *Who can donate to O+?* O+ and O-. The observed output is that anyone with blood group O can donate. This is the same as the expected output.
  - *Who can donate to O-?* O-. The observed output is that anyone with blood group O can donate, with the exception of those who are Rh positive. This is the same as the expected output.
  - *Who can donate to B+?* O+, O-, B+ and B-. The observed output is that anyone with blood group B or O can donate. This is the same as the expected output.
  - *Who can donate to B-?* O- and B-. The observed output is that anyone with Rh factor negative can donate, except when their blood group is A or AB. This is the same as the expected output.
  - *Who can donate to A+?* O+, O-, A+ and A-. The observed output is that anyone with blood group A or O can donate. This is the same as the expected output.
  - *Who can donate to A-?* O- and A-. The observed output is that anyone with Rh factor negative can donate, except when their blood group is B or AB. This is the same as the expected output.
  - *Who can donate to AB+?* Anyone. The observed output is that anyone with blood group A, or Rh factor positive, or Rh factor negative can donate. Although this output is correct, it is not minimal. Merely the rule "donatestoAB- :- True." would have sufficed.
  - *Who can donate to AB-?* O-, A-, B- and AB-. The observed output is that anyone with Rh factor negative can donate. This is the same as the expected output.

3.3.5 Example 5 - Nested exception

- **Input:** Patient1 has blood cancer. Someone has donated marrow to Patient 1. The marrow that Patient1 has received is incompatible. Patient2 has blood cancer. Patient3 has blood cancer. Someone has donated marrow to Patient3. Patient1 and Patient2 die, while Patient3 does not.
- **Expected output:** All patients who have blood cancer will die, except for those who have received a marrow donation. However, even from those who have received a marrow donation, if the marrow is incompatible, they will die.
- **Observed output:** It is the same as the expected output.

3.3.6 Example 6 - Nested exception

- **Input:** Anyone who has graduated gives the interview. Alice, Bob, Charlie and Devon have graduated. Alice, Bob and Charlie have low GPAs. Alice and Bob had a personal difficulty (hence the low GPA). Frank and George give the interview. Frank is brilliant. Alice, Bob, Devon and Frank are selected, while Charlie and George are not.
- **Expected output:** Candidates who have graduated are accepted, except for those who have low GPA. If they had had a personal difficulty, they are accepted. If a brilliant person gives the interview, he is accepted.
- **Observed output:** It is the same as the expected output.

3.4 Mistakes in the algorithm

While implementing the algorithm, I observed that there are a few mistakes in the algorithm presented in the paper.

1. In the "specialize" function, the while loop on line 11 should be a do-while loop, i.e. the steps inside it should be executed at least once before the check is performed.

2. In the "specialize" function, the statement on line 27 updates the negative examples to those that were covered in the previous rule, but are not covered by the updated rule. This is wrong. The negative examples should be updated to those that are covered by the updated rule.

3. In order to obtain correct results as per the definition of information gain described in Section 2.4, sometimes a clause has to be added to a rule even though the information gain is 0. Therefore the lines 13 and 32 in the "specialize" and "exception" functions respectively need to be changed to check whether the rule returned by "addBestLiteral" is NULL.

I have incorporated these changes in the implementation of the algorithm.

4 Project Source Code

My implementation of the FOLD algorithm is available at:
https://github.com/cpethe/ComputingWithLogic/blob/master/FOLD%20algorithm.ipynb

5 Conclusion

The FOLD algorithm learns rules with exceptions very well. In one test example, the output was correct but not minimal. In all other cases, the rules learned by the algorithm are perfectly the same as those which a human would describe intuitively.

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] Stephen Muggleton, *Inductive Logic Programming*, New Generation Computing, 8, 295-318, 1991.