TLP **17** (5–6): 1010–1026, 2017. © Cambridge University Press 2017
doi:10.1017/S1471068417000333 First published online 23 August 2017

*A new algorithm to automate inductive learning of default theories**

FARHAD SHAKERIN, ELMER SALAZAR and GOPAL GUPTA
The University of Texas at Dallas, Texas, USA
(*e-mails:* fxs130430@utdallas.edu, ees101020@utdallas.edu, gupta@utdallas.edu)

submitted 14 July 2017; revised 8 August 2017; accepted 27 July 2017

Abstract

In inductive learning of a broad concept, an algorithm should be able to distinguish concept examples from exceptions and noisy data. An approach through recursively finding patterns in exceptions turns out to correspond to the problem of learning default theories. Default logic is what humans employ in common-sense reasoning. Therefore, learned default theories are better understood by humans. In this paper, we present new algorithms to learn default theories in the form of non-monotonic logic programs. Experiments reported in this paper show that our algorithms are a significant improvement over traditional approaches based on inductive logic programming. Under consideration for acceptance in TPLP.

KEYWORDS: inductive logic programming, non-monotonic logic programming, default reasoning, common-sense reasoning, machine learning

1 Introduction

Predictive models produced by classical machine learning methods are not comprehensible for humans because they are algebraic solutions to optimization problems such as risk minimization or data likelihood maximization. These methods do not produce any intuitive description of the learned model. This makes it hard for users to understand and verify the underlying rules that govern the model. As a result, these methods do not produce any justification when they are applied to a new data sample. Also, extending the prior knowledge[1] in these methods requires the entire model to be re-learned. Additionally, no distinction is made between *exceptions* and noisy data. *Inductive Logic Programming* (Muggleton 1991), however, is one technique where the learned model is in the form of logic programming rules (Horn clauses) that are more comprehensible and that allows the background knowledge to be incrementally extended without requiring the entire model to be relearned.

* Authors are partially supported by NSF Grant No. 1423419.
[1] In the rest of the paper, we will use the term background knowledge to refer to prior knowledge (Muggleton 1991).

This comprehensibility of symbolic rules makes it easier for users to understand and verify the resulting model and even edit the learned knowledge.

Given the background knowledge and a set of positive and negative examples, Inductive Logic Programming (ILP) learns theories in the form of Horn logic programs. However, due to the lack of negation-as-failure (NAF), Horn clauses are not sufficiently expressive for representation and reasoning when the background knowledge is incomplete.

Additionally, ILP is not able to handle exception to general rules: It learns rules under the assumption that there are no exceptions to them. This results in exceptions and noise being treated in the same manner. Often, the exceptions to the rules themselves follow a pattern, and these exceptions can be learned as well. The resulting theory that is learned is a default theory, and in most cases, this theory describes the underlying model more accurately. It should be noted that default theories closely model common sense reasoning as well (Baral 2003). Thus, a default theory, if it can be learned, will be more intuitive and comprehensible for humans. Default reasoning also allows us to reason in absence of information. A system that can learn default theories can therefore learn rules that can draw conclusions based on the lack of evidence, just like humans. Other reasons that underscore the importance of inductive learning of default theories can be found in the paper by Sakama (2005) who also surveys other attempts in this direction.

As an example, suppose we want to learn the concept of flying ability of birds. We would like to learn the default rule that birds normally fly, as well as rules that capture exceptions, namely, penguins and ostriches are birds that do not fly. Current ILP systems will be thrown off by the exceptions and will not discover any general rule: They will just either enumerate all the birds that fly or cover the positive examples without caring much about the falsely covered negative examples. Other algorithms, such as First Order Inductive Learner (FOIL), will induce rules that are non-constructive and thus not helpful or intuitive.

In this paper, we present two algorithms for learning default theories (i.e., non-monotonic logic programs), called First Order Learner of Default (FOLD) and FOLD-R, to handle categorical and numeric features, respectively. Unlike traditional ILP systems that learn standard logic programs (i.e., no negation is allowed), our algorithms learn *non-monotonic stratified logic programs* (that allow *NAF*). Our algorithms are an extension of the FOIL algorithm by Quinlan (1990) and support both categorical and numeric features. Also, the FOLD and FOLD-R learning algorithms can learn recursive rules. Whenever needed, our algorithms introduce new predicates. The language bias (Mitchell 1980) also contains arithmetic constraints of the form $\{A \leqslant h, A \geqslant h\}$. The algorithms have been implemented and tried on variety of datasets from the UCI repository. They have shown excellent results that are presented here as well.

The default theories that we learn using our algorithm, as well as the background knowledge used, are assumed to follow the stable model semantics[2]. Stable model

[2] We assume that the background knowledge has exactly one stable model.

semantics, and its realization in answer set programming (ASP), provides an elegant mechanism for handling negation in logic programming (Gelfond and Lifschitz 1988). We assume that the reader is familiar with ASP and stable model semantics (Baral 2003).

This paper makes the following contributions: We propose a novel concrete algorithm to learn default theories automatically in the absence of complete information. The proposed algorithm, unlike the existing ones, is able to handle the numeric features without discretizing them first, and is also capable of handling non-monotonic background knowledge. We provide both qualitative and quantitative results from standard UCI datasets to support the claim that our algorithm discovers more accurate as well as more intuitive rules compared to the conventional ILP systems.

Rest of the paper is organized as follows: Section 2 formally defines the problem we tackle in this paper. Section 3 presents some background materials. Section 4 presents the FOLD algorithm to solve the problem. In Section 5, we extend FOLD to handle numeric features. Section 6 presents the experiments and results. Section 7 discusses related research. Section 8 discusses our future research direction and finally in Section 9 we conclude.

2 The inductive learning problem

The problem that we tackle in this paper is an inductive non-monotonic logic programming problem that can be formalized as follows.

Given

- a background theory $\mathscr{B}$, in the form of a normal logic program, i.e, clauses of the form $h \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n$, where h and $l_1, \ldots, l_n$ are positive literals and *not* denotes *NAF* with stable model semantics;
- two disjoint sets of grounded goal predicates $\mathscr{E}^+$,$\mathscr{E}^-$ are known as positive and negative examples, respectively;
- a hypothesis language of predicates $\mathscr{L}$ including function and atom free predicates. It also contains a set of arithmetic constraints of the form $\{A \leqslant h, A \geqslant h\}$ where A is a variable and h is a real number;
- a $covers(\mathscr{H}, \mathscr{E}, \mathscr{B})$ function, which returns the subset of $\mathscr{E}$ that is extensionally implied by the current hypothesis $\mathscr{H}$ given the background knowledge $\mathscr{B}$;
- a $score(\mathscr{E}^+, \mathscr{E}^-, \mathscr{H}, \mathscr{B})$ function, which specifies the quality of the hypothesis $\mathscr{H}$ with respect to $\mathscr{E}^+, \mathscr{E}^-, \mathscr{B}$;

Find

- a theory $\mathscr{T}$ for which $covers(\mathscr{T}, \mathscr{E}^+, \mathscr{B}) = \mathscr{E}^+$ and $covers(\mathscr{T}, \mathscr{E}^-, \mathscr{B}) = \emptyset$.

3 Background

Our algorithm to learn default theories is an extension of the FOIL algorithm (Quinlan 1990). FOIL is a top-down ILP system that follows a *sequential covering* approach to induce a hypothesis. The FOIL algorithm is summarized in Algorithm

1. This algorithm repeatedly searches for clauses that score best with respect to a subset of positive and negative examples, a current hypothesis and a heuristic called *information gain* (IG).

Algorithm 1 Summarizing the FOIL algorithm

Input: $goal, \mathscr{B}, \mathscr{E}^+, \mathscr{E}^-$
Output: Initialize $\mathscr{H} \leftarrow \emptyset$
1: **while not**(*stopping criterion*) **do**
2: $c \leftarrow (goal$:- $true.)$
3: **while not**(*stopping criterion*) **do**
4: **for** all $c' \in \rho(c)$ **do**
5: *compute score*$(\mathscr{E}^+, \mathscr{E}^-, \mathscr{H} \cup \{c'\}, \mathscr{B})$
6: **end for**
7: let $\hat{c}$ be the $c' \in \rho(c)$ with the best score
8: $c \leftarrow \hat{c}$
9: **end while**
10: add $\hat{c}$ to $\mathscr{H}$
11: $\mathscr{E}^+ \leftarrow \mathscr{E}^+ \setminus covers(\hat{c}, \mathscr{E}^+, \mathscr{B})$
12: **end while**

The inner loop searches for a clause with the highest IG using a general-to-specific hill-climbing search. To specialize a given clause c, a refinement operator ρ under θ-subsumption (Plotkin 1971) is employed. The most general clause is $p(X_1, \ldots, X_n) \leftarrow true$, where the predicate p/n is the predicate being learned and each X_i is a variable. The refinement operator specializes the current clause $h \leftarrow b_1, \ldots b_n$. This is realized by adding a new literal l to the clause yielding $h \leftarrow b_1, \ldots b_n, l$. The heuristic-based search uses IG. In FOIL, IG for a given clause is calculated as follows (Mitchell 1997):

$$IG(L, R) = t\left(log_2 \frac{p_1}{p_1 + n_1} - log_2 \frac{p_0}{p_0 + n_0}\right) \tag{3.1}$$

where 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 $R + L$ and t is the number of positive examples implied by R also covered by $R + L$. FOIL handles negated literals in a naive way by adding the literal *not* L to the set of specialization candidate literals for any existing candidate L. This approach leads to learning predicates that do not capture the concept accurately as shown in the following example.

Example 3.1

$\mathscr{B}, \mathscr{E}^+$ are background knowledge and positive examples, respectively, with Closed-World Assumption (CWA) and the concept to be learned is fly:

$$\begin{array}{ll} \mathscr{B}: bird(X) \leftarrow penguin(X). & \\ \quad bird(tweety). & bird(et). \\ \quad cat(kitty). & penguin(polly). \\ \mathscr{E}^+: fly(tweety). & fly(et). \end{array}$$

The FOIL algorithm would learn the following rule:

$$fly(X) \leftarrow not\ cat(X), not\ penguin(X).$$

Although this rule covers all the positives (tweety and et are not penguins and cats) and no negatives (kitty and polly do not satisfy the clause body), it still does not yield an intuitive rule. In fact, the correct theory in this example is as follows: "*Only birds fly but, among them there are exceptional ones who do not fly*". It translates to the following Prolog rule:

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

which FOIL fails to discover.

4 FOLD algorithm

The idea of our FOLD algorithm is to learn a concept as a default theory and possibly multiple exceptions. In that sense, FOLD tries first to learn the default by specializing a general rule of the form $goal(V_1, \ldots, V_n) \leftarrow true.$ with positive literals. As in FOIL, each specialization must rule out some already covered negative examples without decreasing the number of positive examples covered significantly. Unlike FOIL, no negative literal is used at this stage. Once the IG becomes zero, this process stops. At this point, if some negative examples are still covered, they must be either noisy data samples or exceptions to the so far learned rule. As Srinivasan *et al.* (1996) discuss, there is no pattern distinguishable in noise, whereas, in exceptions, there may exist a pattern that can be described using the same language bias. This can be viewed as a subproblem to (recursively) find the rules governing a set of negative examples. To achieve that aim, FOLD swaps the current positive and negative examples and recursively calls the FOLD algorithm to learn the exception rule(s). Each time a rule is discovered for exceptions, a new predicate $ab(V_1, \ldots, V_n)$ is introduced. To avoid name collision, FOLD appends a unique number at the end of the string *ab* to guarantee the uniqueness of the invented predicates.

In the case of noisy data or in the presence of uncertainty due to the lack of information, it turns out that there is no pattern to learn. In such cases, FOLD enumerates the positive examples for two purposes: first, this is essential for the training algorithm to converge, and second, it helps to detect noisy data samples.

Algorithm 2 shows a high-level implementation of the FOLD algorithm. In lines 1–8, function FOLD serves as the FOIL outer loop. In line 3, FOLD starts with the most general clause (e.g. $fly(X) \leftarrow true.$). In line 4, this clause is refined by calling the function $SPECIALIZE$. In lines 5 and 6, set of positive examples and set of discovered clauses are updated to reflect the newly discovered clause.

In lines 9–29, the function $SPECIALIZE$ is shown. It serves as the FOIL inner loop. In line 12, by calling the function ADD_BEST_LITERAL, the "best" positive literal is chosen and the best IG as well as the corresponding clause is returned. In lines 13–24, depending on the IG value, either the positive literal is accepted or the EXCEPTION function is called. If, at the very first iteration, IG becomes zero, then a clause that just enumerates the positive examples is produced. A flag called *just_started* is used to differentiate the first iteration. In lines 26–27, the sets of positive and negative examples are updated to reflect the changes of the current clause. In line 19, the EXCEPTION function is called while swapping the $\mathscr{E}^+, \mathscr{E}^-$.

In line 31, we find the "best" positive literal that covers more positive examples and fewer negative examples. Again, note that the current positive examples are really the negative examples and in the EXCEPTION function, we try to find the rule(s) governing the exception. In line 33, FOLD is recursively called to extract this rule(s). In line 34, a new *ab* predicate is introduced and in lines 35–36 it is associated with the body of the rule(s) found by the recurring FOLD function call in line 33. Finally, in line 38, default and exception are attached together to form a single clause.

The FOLD algorithm, once applied to Example 3.1, yields the following clauses:

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

Now, we illustrate how FOLD discovers the above set of clauses given $\mathscr{E}^+ = \{tweety, et\}$ and $\mathscr{E}^- = \{polly, kitty\}$ and the goal $fly(X)$. By calling FOLD, in line 2 "while", the clause $fly(X) \leftarrow true.$ is specialized. In the $SPECIALIZE$ function, in line 12, the literal $bird(X)$ is picked to add to the current clause, to get the clause $\hat{c} = fly(X) \leftarrow bird(X)$ that happened to have the greatest IG among $\{bird, penguin, cat\}$. Then, in line 26–27, the following updates are performed: $\mathscr{E}^+ = \{\}$, $\mathscr{E}^- = \{polly\}$. A negative example *polly*, a penguin is still covered. In the next iteration, $SPECIALIZE$ fails to introduce a positive literal to rule it out since the best IG in this case is zero. Therefore, the EXCEPTION function is called by swapping the $\mathscr{E}^+$, $\mathscr{E}^-$. Now, FOLD is recursively called to learn a rule for $\mathscr{E}^+ = \{polly\}$, $\mathscr{E}^- = \{\}$. The recursive call (line 33), returns $fly(X) \leftarrow penguin(X)$ as the exception. In line 34, a new predicate $ab0$ is introduced and in line 35–37 the clause $ab0(X) \leftarrow penguin(X)$ is created and added to the set of invented abnormalities namely, AB. In line 38, the negated exception (i.e., not $ab0(X)$) and the default rule's body (i.e., $bird(X)$) are compiled together to form the clause $fly(X) \leftarrow bird(X), not\ ab0(X)$.

Note, in two different cases *enumerate* is called. First, at very first iteration of specialization if IG is zero for all the positive literals. Second, when the *Exception* routine fails to find a rule governing the negative examples. Whichever is the case, corresponding samples are considered as noise. The following example shows a learned logic program in the presence of noise. In particular, it shows how *enumerate* function in FOLD works: It generates clauses in which the variables of the goal predicate can be unified with each member of a list of the examples for which no pattern exists.

Algorithm 2 FOLD algorithm

Input: $goal, \mathscr{B}, \mathscr{E}^+, \mathscr{E}^-$
Output:
$D = \{c_1, \ldots, c_n\}$ ▷ defaults' clauses
$AB = \{ab_1, \ldots, ab_m\}$ ▷ exceptions/abnormal clauses

1: **function** FOLD($\mathscr{E}^+, \mathscr{E}^-$)
2: **while** ($size(\mathscr{E}^+) > 0$) **do**
3: $c \leftarrow (goal$:- $true.)$
4: $\hat{c} \leftarrow$ SPECIALIZE($c, \mathscr{E}^+, \mathscr{E}^-$)
5: $\mathscr{E}^+ \leftarrow \mathscr{E}^+ \setminus covers(\hat{c}, \mathscr{E}^+, \mathscr{B})$
6: $D \leftarrow D \cup \{\hat{c}\}$
7: **end while**
8: **end function**
9: **function** SPECIALIZE($c, \mathscr{E}^+, \mathscr{E}^-$)
10: $just\_started \leftarrow true$
11: **while** ($size(\mathscr{E}^-) > 0$) **do**
12: $(c_{def}, \hat{IG}) \leftarrow$ ADD_BEST_LITERAL($c, \mathscr{E}^+, \mathscr{E}^-$)
13: **if** $\hat{IG} > 0$ **then**
14: $\hat{c} \leftarrow c_{def}$
15: **else**
16: **if** $just\_started$ **then**
17: $\hat{c} \leftarrow enumerate(c, \mathscr{E}^+)$
18: **else**
19: $\hat{c} \leftarrow$ EXCEPTION($c, \mathscr{E}^-, \mathscr{E}^+$)
20: **if** $\hat{c} = null$ **then**
21: $\hat{c} \leftarrow enumerate(c, \mathscr{E}^+)$
22: **end if**
23: **end if**
24: **end if**
25: $just\_started \leftarrow false$
26: $\mathscr{E}^+ \leftarrow \mathscr{E}^+ \setminus covers(\hat{c}, \mathscr{E}^+, \mathscr{B})$
27: $\mathscr{E}^- \leftarrow \mathscr{E}^- \setminus covers(\hat{c}, \mathscr{E}^-, \mathscr{B})$
28: **end while**
29: **end function**
30: **function** EXCEPTION($c_{def}, \mathscr{E}^+, \mathscr{E}^-$)
31: $\hat{IG} \leftarrow$ ADD_BEST_LITERAL($c, \mathscr{E}^+, \mathscr{E}^-$)
32: **if** $\hat{IG} > 0$ **then**
33: $c\_set \leftarrow$ FOLD($\mathscr{E}^+, \mathscr{E}^-$)
34: $c\_ab \leftarrow generate\_next\_ab\_predicate()$
35: **for each** $c \in c\_set$ **do**
36: $AB \leftarrow AB \cup \{c\_ab$:- $bodyof(c)\}$
37: **end for**
38: $\hat{c} \leftarrow (headof(c_{def})$:- $bodyof(c)$, **not**$(c\_ab))$
39: **else**
40: $\hat{c} \leftarrow null$
41: **end if**
42: **end function**

Example 4.1

Similar to Example 3.1, plus we have an extra positive example fly(jet) without any further information:

$$\begin{aligned}\mathscr{B}: & bird(X) \leftarrow penguin(X).\\ & bird(tweety). \qquad bird(et).\\ & cat(kitty). \qquad penguin(polly).\\ \mathscr{E}^+: & fly(tweety). \quad fly(jet). fly(et).\end{aligned}$$

The FOLD algorithm on the Example 4.1 yields the following clauses:

$$\begin{aligned}&fly(X) \leftarrow bird(X), not\ ab0(X).\\ &fly(X) \leftarrow member(X, [jet]).\\ &ab0(X) \leftarrow penguin(X).\end{aligned}$$

FOLD recognizes *jet* as a noisy data. $member/2$ is a built-in predicate in SWI-Prolog to test the membership of an atom in a list.

Sometimes, there are nested levels of exceptions. The following example shows how FOLD manages to learn the correct theory in presence of nested exceptions.

Example 4.2

Birds and planes normally fly, except penguins and damaged planes that cannot. There are super penguins who can, exceptionally, fly:

$$\begin{aligned}\mathscr{B}: & bird(X) \leftarrow penguin(X).\\ & penguin(X) \leftarrow superpenguin(X).\\ & bird(a). \quad bird(b). \quad penguin(c). \quad penguin(d).\\ & superpenguin(e). \quad superpenguin(f). \quad cat(c1).\\ & plane(g). \quad plane(h). \quad plane(k). \quad plane(m).\\ & damaged(k). \quad damaged(m).\\ \mathscr{E}^+: & fly(a). \quad fly(b). \quad fly(e).\\ & fly(f). \quad fly(g). \quad fly(h).\end{aligned}$$

The FOLD algorithm learns the following theory:

$$\begin{aligned}&fly(X) \leftarrow plane(X), not\ ab0(X).\\ &fly(X) \leftarrow bird(X), not\ ab1(X).\\ &fly(X) \leftarrow superpenguin(X).\\ &ab0(X) \leftarrow damaged(X).\\ &ab1(X) \leftarrow penguin(X).\end{aligned}$$

Theorem 1

The FOLD algorithm terminates on any finite set of examples.

Proof

It suffices to show that the size of $\mathscr{E}^+$ on every iteration of the FOLD function decreases (at line 5) and since $\mathscr{E}^+$ is a finite set, it will eventually become empty and the while loop terminates. Equivalently, we can show that every time the SPECIALIZE function is called, it terminates and a clause $\hat{c}$ that covers a non-empty subset of $\mathscr{E}^+$ is returned. Inside the SPECIALIZE function, if $\mathscr{E}^-$ is empty, then the function returns its input clause and the theorem trivially holds. Otherwise, two cases might happen: First, SPECIALIZE produces a clause that enumerates $\mathscr{E}^+$ and covers no negative example. In such a case, it returns immediately and the theorem trivially holds.

Second, SPECIALIZE calls the EXCEPTION function that may lead to a chain of recursive calls on FOLD function. In this case, it suffices to show that on a chain of recursive calls on FOLD, the size of function argument, i.e., $\mathscr{E}^+$ decreases each time a new call to FOLD occurs. That is indeed the case because every time a literal is added to the current clause at line 12, it covers fewer negative examples from $\mathscr{E}^-$ that in turn becomes the new $\mathscr{E}^+$ as the EXCEPTION function and subsequently the FOLD function is recursively called. Therefore, on consecutive calls to FOLD function, the size of input argument $\mathscr{E}^+$ is decreased until it eventually terminates. □

Theorem 2
The FOLD algorithm always learns a hypothesis that covers no negative example (soundness).

Proof
It follows from the Theorem 1 that every loop in the algorithm (and, subsequently the algorithm) terminates. In particular, the while loop inside the function SPECIALIZE terminates as soon as the negated loop condition (i.e., the number of negative examples covered by the rule being discovered equals zero) starts to hold. Since, for every learned rule, no negative example is covered, it follows that the FOLD algorithm learns a hypothesis that covers no negative example. □

Theorem 3
The FOLD algorithm always learns a hypothesis that covers all positive examples (completeness).

Proof
The proof is similar to the soundness proof. □

5 Numeric extension of FOLD

ILP systems have limited application to datasets containing a mix of categorical and numerical features. A common way to deal with numerical features is to discretize the data to qualitative values. This approach leads to accuracy loss and requires domain expertise. Instead, we adapt the approach taken in the well-known C4.5 algorithm (Quinlan 1993). This algorithm is ranked no. 1 in the survey paper "*Top 10 algorithms in datamining*" by Wu *et al.* (2007). For a numeric feature A, constraints such as $\{A \leqslant h, A > h\}$ have to be considered where the threshold h is found by sorting the values of A and choosing the split between successive values that maximizes the IG. In our FOLD-R algorithm that we propose and describe next, we perform the same method for a set of operators $\{<, \leqslant\}$ and pick the operator and threshold that maximizes the IG. Also, we need to extend the ILP language bias to support the arithmetic constraints.

Unlike the categorical features for which we use *propositionalization* (Kramer *et al.* 2000), for numerical features, we define a predicate that contains an extra variable that always pairs with a constraint. For example to extend the language bias for a numeric quantity "age", we could define predicates of the form $age(a, b)$ in the background knowledge, and the candidate to specialize a clause might be as

follows: $age(X, N), N \leqslant 5$. However, the predicate $age/2$ never appears without the corresponding constraint.

Algorithm 3 illustrates the high-level changes made to FOLD, in order to obtain the FOLD-R algorithm. The function *test_categorical*, as before, chooses the best categorical literal to specialize the current clause. The function *test_numeric* chooses the best numeric literal as well as the best arithmetic constraint and threshold with the highest IG. In line 5, if neither one leads to a positive IG, EXCEPTION is tried. If exception also fails, then *enumerate* is called. Otherwise, IGs are compared and whichever is greater, the corresponding clause is chosen as the specialized clause of the current iteration.

Algorithm 3 FOLD-R algorithm, specialize function. Other functions are the same as in FOLD

```
1:  function SPECIALIZE(c, E+, E−)
2:      while (size(E−) > 0) do
3:          (ĉ1, IĜ1) ← test_categorical(c, E+, E−)
4:          (ĉ2, IĜ2) ← test_numeric(c, E+, E−)
5:          if IĜ1 = 0 & IĜ2 = 0 then
6:              ĉ ← EXCEPTION(c, E−, E+)
7:              if ĉ = null then
8:                  ĉ ← enumerate(c, E+)
9:              end if
10:         else
11:             if IĜ1 ⩾ IĜ2 then
12:                 ĉ ← ĉ1
13:             else
14:                 ĉ ← ĉ2
15:             end if
16:         end if
17:         E+ ← E+ \ covers(ĉ, E+, B)
18:         E− ← E− \ covers(ĉ, E−, B)
19:     end while
20: end function
```

Example 5.1

Table 1 adapted from Quinlan (1993) is a dataset with numeric features "temperature" and "humidity". "Outlook" and "Windy" are categorical features. Our FOLD-R algorithm, for the goal $play(X)$, and positive examples shown as records with label "Play", and negative examples shown as records with label "Don't Play" outputs the following clauses:

$$play(X) \leftarrow overcast(X).$$
$$play(X) \leftarrow temperature(X, A), A \leqslant 75, not\ ab0(X).$$
$$ab0(X) \leftarrow windy(X), rainy(X).$$
$$ab0(X) \leftarrow humidity(X, A), A \geqslant 95, sunny(X).$$

Table 1. *Play Tennis dataset, numeric version*

Outlook	Temperature	Humidity	Wind	PlayTennis
Sunny	75	70	True	Play
Sunny	80	90	True	Don't Play
Sunny	85	85	False	Don't Play
Sunny	72	95	False	Don't Play
Sunny	69	70	False	Play
Overcast	72	90	True	Play
Overcast	83	78	False	Play
Overcast	83	65	True	Play
Overcast	81	75	False	Play
Rain	71	80	True	Don't Play
Rain	65	70	True	Don't Play
Rain	75	80	False	Play
Rain	68	80	False	Play
Rain	70	96	False	Play

FOLD-R results suggest an abnormal day to play is either a rainy and windy day or a sunny day with humidity above 95%.

6 Experiments and results

This section presents the results obtained with the FOLD-R algorithm on some of the standard UCI datasets. To conduct the following experiments, we implemented the algorithm in Java. We used Prolog queries to process the background knowledge (the background knowledge is assumed to be represented as a standard Prolog program). For performing IG computations and CWA generation of negative examples, we made the use of the JPL library (Singleton and Dushin 2003) that interfaces SWI-Prolog version 7.1.23-34 (Wielemaker *et al.* 2012) with Java. Our intention here is to investigate the quality of discovered rules both in terms of their accuracy and the degree to which they are consistent with the common sense understanding from the underlying concepts. To measure the accuracy, we implemented 10-fold cross-validation on each dataset and the mean of calculated accuracy is represented, while the standard deviation for all the datasets were 5% or lower. At present, we are not greatly interested in the running time and/or space complexity of the algorithm: This will be subject of future research. All the learning tasks were performed using a PC with Intel(R) Core(TM) i7-4700HQ CPU @ 2.40 GHz and 8.00 GB RAM. Execution times ranges from a few seconds to a few minutes. The bottleneck is the function that sorts the numeric values to pick the best threshold and operator. There are solutions to get around this such as those proposed by Catlett (1991). Table 2 reports the execution time of FOLD-R on our benchmarks. The algorithm works much faster when no numeric feature is present in the dataset. It should be noted that calls to JPL add significant overhead to the algorithm's execution time.

Labor Relations: The data include a set of contracts that depending on their features (16 features) are classified as good or bad contracts. The following set of clauses for a good contract are discovered by FOLD-R:

$$good\_cont(X) \leftarrow wage\_inc\_f(X, A), A > 2, not\ ab0(X).$$
$$good\_cont(X) \leftarrow holidays(X, A), A > 11.$$
$$good\_cont(X) \leftarrow hplan\_half(X), pension(X).$$
$$ab0(X) \leftarrow no\_long\_disability\_help(X).$$
$$ab0(X) \leftarrow no\_pension(X).$$

According to the first rule, a contract with 2% wage increase (default) is a good contract except when the employer does not contribute in a possible long-term disability and a pension plan. According to the second rule, a contract with holidays above 11 days is also good. And, finally, if employer contributes in half of the health plan and entire pension plan, the contract is good.

Mushroom: This dataset includes descriptions of different species of mushrooms and their features that are used to classify whether they are poisonous or edible. The following set of clauses for a poisonous mushroom is discovered by FOLD:

$$poisonous(X) \leftarrow ring\_type\_none(X).$$
$$poisonous(X) \leftarrow spore\_print\_color\_green(X).$$
$$poisonous(X) \leftarrow gill\_size\_narrow(X), not\ ab2(X).$$
$$poisonous(X) \leftarrow odor\_foul(X).$$
$$ab0(X) \leftarrow population\_clustered(X).$$
$$ab0(X) \leftarrow stalk\_surface\_below\_ring\_scaly(X).$$
$$ab1(X) \leftarrow stalk\_shape\_enlarging(X).$$
$$ab2(X) \leftarrow gill\_spacing\_crowded(X), not\ ab1(X).$$
$$ab2(X) \leftarrow odor\_none(X), not\ ab0(X).$$

Note, the induced theory has nested exceptions. This nesting happens as a result of finding patterns for negative examples, which makes the FOLD algorithm perform more recursive steps until no covered negative example is left.

Table 2 compares the accuracy of FOLD-R algorithm against that of ALEPH by Srinivasan (2001). The examples have been picked from well-known standard datasets for some of which ALEPH exhibits low test accuracy. In most cases, FOLD-R accuracy outperforms ALEPH. The experiments suggest that when the absence of a particular feature value plays a crucial role in classification, our algorithm shows a meaningful higher accuracy. This follows from the fact that the classical ILP algorithms only make use of existent information as opposed to *NAF* in which a decision is made based on the absence of information. As an example, in the credit-j dataset, our algorithm generates four rules with abnormality predicates. These rules cover positive examples that without abnormality predicates would have remained uncovered. However, in Bridges and Ecoli where no abnormality predicate is introduced by our algorithm, both ALEPH and FOLD-R end-up with almost the same accuracy.

Even in cases where no improvement over accuracy is achieved, our default theory approach leads to simpler and more intuitive rules. As an example, in the case of Mushroom, other ILP systems, including ALEPH and FOIL, would produce nine

Table 2. *FOLD-R evaluation on UCI benchmarks*

Dataset	Size	ALEPH accuracy(%)	FOLD-R accuracy(%)	FOLD-R execution time(s)
Credit-au	690	82	83	67
Credit-j	125	53	81	20
Credit-g	1,000	70.9	78	87
Iris	150	85.9	95	1.3
Ecoli	336	91	90	6.1
Bridges	108	89	90	0.8
Labor	57	89	94	0.4
Acute(1)	34	100	100	0.3
Acute(2)	34	100	100	0.3
Mushroom	7,724	100	100	11.4

rules with two literals each in the body to cover all the positives, while our FOLD algorithm, produces three single-literal rules and one rule with two literals in which the second literal takes care of the exceptions.

7 Related work

Sakama (2005) discusses the necessity of having a non-monotonic language bias to perform induction for default reasoning. It surveys some of the proposals directly adapted from ILP, like *inverse resolution* (Muggleton and Buntine 1988) and *inverse entailment* (Muggleton 1995) and then he explains why these are not applicable to non-monotonic logic programs. Sakama then introduces an algorithm to induce rules from *answer sets* that generalizes a rule from specific grounded rules in a bottom-up fashion. His approach in some cases yields premature generalizations that produces redundant negative literals in the body of the rule and therefore over-fitted to the training data. The following example illustrates what Sakama's algorithm would produce:

Example 7.1

$$\mathscr{B}: bird(X) \leftarrow penguin(X).$$
$$bird(tweety). \qquad bird(et).$$
$$bear(teddy). \qquad crippled(et).$$
$$cat(kitty). \qquad penguin(polly).$$
$$\mathscr{E}^{+}: fly(tweety).$$

and the algorithm outputs the following rule:

$$fly(X) \leftarrow bird(X), not\ cat(X), not\ penguin(X), not\ bear(X), not\ crippled(X).$$

in which some of the literals including not cat(X) and not bear(X) are redundant.

Additionally, since ASP systems have to ground the predicates to produce the answer set, introducing numeric data in background knowledge and also in the language bias is prohibited.

In a different line of research, Dimopoulos and Kakas (1995) describe an algorithm to learn exception using the patterns in the negative examples. However, they do

not make any use of NAF as the core notion of reasoning in the absence of complete information. Instead, their algorithm learns a hierarchical logic program, including classical negation, in which the order of rules prioritize their application and, therefore, it is not naturally compatible with standard Prolog.

The idea of swapping positive and negative examples to learn patterns from negative examples has first been discussed by Srinivasan *et al.* (1996) where a bottom-up ILP algorithm is proposed to specialize a clause after it has already been generalized and still covers negative examples. Similarly, Inoue and Kudoh (1997) propose a bottom-up algorithm in two phases: First, producing monotonic rules via a standard ILP method, and then specializing them by introducing negated literals to the body of the rule. In contrast, we believe our FOLD algorithm that takes a top-down approach learns programs that have a better fit, thanks to its support for numeric features and its better scalability. The lacks of both are inherent problems in bottom-up ILP methods.

ALEPH (Srinivasan 2001) is one of the most widely used ILP systems that uses a bottom-up generalization search to induce theories that cover the maximum possible positive examples. However, since the induced theory might be overly generalized, there is an option to refine the theory by introducing abnormality predicates that rule out negative examples by specializing an overly generalized rule. This specialization step is manual and unlike our algorithm, no automation is offered by ALEPH. Also, ALEPH does not support numeric features.

XHAIL (Ray 2009) is another non-monotonic ILP framework that integrates abductive, deductive and inductive forms of reasoning to form sets of ground clauses called kernel set and then generalize it to learn non-stratified logic programs.

In a different line of research, Corapi *et al.* (2012) approaches the non-monotonic ILP problem by incorporating the power of modern ASP solvers to search for an optimal hypothesis among the generated so-called skeleton rules and a set of abducibles associated to them. The last two approaches do not scale up as the language bias grows.

8 Future work

One advantage of our FOLD-R algorithm over the existing systems is the ability to handle non-monotonic background knowledge. Conventional ILP systems permit only standard Prolog programs to represent background knowledge. In contrast, our FOLD-R algorithm, once integrated with a top down *ASP* system like s(ASP) (Marple and Gupta 2012), will permit the background knowledge to be represented as an answer set program. The idea of leveraging the added expressiveness of non-monotonic logic-based background knowledge has been discussed by Seitzer (1997). However, because Seitzer's method is based on grounding the background knowledge and then computing its answer sets, it is not scalable. Using a query-driven system allows our learning algorithm to be scalable. Extending our algorithms to allow non-monotonic logic-based background knowledge with multiple stable models is part of the future work.

Our eventual goal is to develop a unified framework for learning default theories: (i) in which we can learn hypotheses that are general answer set programs (i.e., these learned answer set programs may not be stratified), and (ii) that work with background knowledge that may be represented as a non-stratified answer set program. Also, the optimality of the learned hypothesis due to greedy nature of IG heuristic is not guaranteed and changing the search strategy from A* to better algorithms such as iterative deepening search is subject of future research.

9 Conclusion

In this paper, we introduced a new algorithm called FOLD to learn default theories. Next, we proposed FOLD-R that is an extension of FOLD to handle numeric features. Both the FOLD and the FOLD-R learning algorithms learn stratified answer set programs that allow recursion through positive literals. Our experiments based on using the standard UCI benchmarks suggest that the default theory, in most cases, describes the underlying model more accurately compared to conventional ILP systems. Default theories also happen to closely model common-sense reasoning. Thus, rules learned from FOLD and FOLD-R are more intuitive and comprehensible by humans. Unlike classical machine learning approaches that learn based on existing information, FOLD and FOLD-R are able to find patterns of information that is absent and express it via a default theory, i.e., as a non-monotonic logic program.

References

Albert, E., Ed. 2013. *Logic-Based Program Synthesis and Transformation, 22nd International Symposium, LOPSTR 2012*, Revised Selected Papers. Lecture Notes in Computer Science, Vol. 7844, September 18–20, 2012. Springer, Leuven, Belgium.

Bain, M. 1991. Experiments in non-monotonic learning. In *Proceedings of the Eighth International Workshop (ML91)*, Northwestern University, Morgan Kaufmann, Evanston, Illinois, USA, 380–384.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, New York, Melbourne.

Birnbaum, L. and Collins, G., Eds. 1991. Experiments in non-monotonic learning. In *Proc. of the 8th International Workshop (ML91)*, Northwestern University, Morgan Kaufmann, Evanston, Illinois, USA.

Catlett, J. 1991. Megainduction: A test flight. In *Proc. of the Eighth International Workshop (ML91)*, Northwestern University, Evanston, Illinois. 596–599.

Corapi, D., Russo, A. and Lupu, E. 2012. *Inductive Logic Programming in Answer Set Programming*. Springer, Berlin, Heidelberg, 91–97.

Dimopoulos, Y. and Kakas, A. C. 1995. Learning non-monotonic logic programs: Learning exceptions. In *Proc. of 8th European Conference on Machine Learning, ECML-95*, Lecture Notes in Computer Science, April 25–27, 1995, Vol. 912. Springer, Heraclion, Crete, Greece, 122–137.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of the 5th International Conference and Symposium*, Vol. 2, August 15–19, 1988. MIT Press, Seattle, Washington, 1070–1080.

Inoue, K. and Kudoh, Y. 1997. Learning extended logic programs. In *Proc. of the 15th International Joint Conference on Artifical Intelligence*, IJCAI'97, Vol. 1. Morgan Kaufmann Publishers, San Francisco, CA, USA, 176–181.

Kowalski, R. A. and Bowen, K. A., Eds. 1988. *Logic Programming*, *Proceedings of the 5th International Conference and Symposium*, Vol. 2, August 15–19, 1988. MIT Press, Seattle, Washington.

Kramer, S., Lavrač, N. and Flach, P. 2000. Propositionalization approaches to relational data mining. In *Relational Data Mining*. Springer-Verlag, New York, NY, USA, 262–286.

Laird, J. E., Ed. 1988. *Machine Learning, Proceedings of the 5th International Conference on Machine Learning*, Ann Arbor, Michigan, USA, June 12–14, 1988. Morgan Kaufmann.

Lavrac, N. and Wrobel, S., Eds. 1995. *Machine Learning*, *Proceedings of 8th European Conference on Machine Learning, ECML-95*, Lecture Notes in Computer Science, April 25–27, 1995, Vol. 912. Springer, Heraclion, Crete, Greece.

Marple, K. and Gupta, G. 2012. Galliwasp: A goal-directed answer set solver. In *Logic-Based Program Synthesis and Transformation. 22nd International Symposium, LOPSTR 2012*, Revised Selected Papers. Lecture Notes in Computer Science, Vol. 7844, September 18–20, 2012. Springer, Leuven, Belgium, 122–136.

Mitchell, T. M. 1980. The need for biases in learning generalizations. In *Readings in Machine Learning*, J. W. Shavlik and T. G. Dietterich, Eds. Morgan Kauffman, 184–191. Book published in 1990.

Mitchell, T. M. 1997. *Machine Learning*. McGraw Hill Series in Computer Science. McGraw-Hill.

Muggleton, S. 1991. Inductive logic programming. *New Generation Computing 8,* 4, 295–318.

Muggleton, S. 1995. Inverse entailment and progol. *New Generation Computing 13,* 3–4, 245–286.

Muggleton, S. and Buntine, W. L. 1988. Machine invention of first order predicates by inverting resolution. In *Machine Learning, Proceedings of the 5th International Conference on Machine Learning*, Ann Arbor, Michigan, USA, June 12–14, 1988. Morgan Kaufmann, 339–352.

Nicolas, P. and Duval, B. 2001. *Representation of Incomplete Knowledge by Induction of Default Theories*. Springer, Berlin, Heidelberg, 160–172.

Plotkin, G. D. 1971. A further note on inductive generalization. In *Machine Intelligence*, Vol. 6. American Elsevier, 101–124.

Quinlan, J. R. 1990. Learning logical definitions from relations. *Machine Learning 5*, 239–266.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Ray, O. 2009. Nonmonotonic abductive inductive learning. *Journal of Applied Logic 7,* 3, 329–340. Special Issue: Abduction and Induction in Artificial Intelligence.

Sakama, C. 2005. Induction from answer sets in nonmonotonic logic programs. *ACM Transactions on Computational Logic 6,* 2, 203–231.

Seitzer, J. 1997. Stable ILP: Exploring the added expressivity of negation in the background knowledge. In *Proc. of IJCAI-97 Workshop on Frontiers of ILP*.

Singleton, P. and Dushin, F. 2003. JPL a java interface to prolog [online]. URL: http://www.swi-prolog.org/packages/jpl/java_api [Accessed on: 26/01/2017].

Srinivasan, A. 2001. Aleph manual [online]. URL: http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/ [Accessed on: 26/01/2017].

Srinivasan, A., Muggleton, S. and Bain, M. 1996. Distinguishing exceptions from noise in non-monotonic learning. In *Proc. of 2nd International Inductive Logic Programming Workshop (ILP'92)*.

Wielemaker, J., Schrijvers, T., Triska, M. and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12*,1–2, 67–96.

Wu, X., Kumar, V., Ross Quinlan, J., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G. J., Ng, A., Liu, B., Yu, P. S., Zhou, Z.-H., Steinbach, M., Hand, D. J. and Steinberg, D. 2007. Top 10 algorithms in data mining. *Knowledge and Information Systems 14,* 1 (December), 1–37.