Cumulative Scoring-based Induction of Default Theories

Farhad Shakerin
The University of Texas at Dallas
[Texas, USA]
fxs130430@utdallas.edu
https://orcid.org/0000-0002-1825-0097

Gopal Gupta
The University of Texas at Dallas
[Texas, USA]
gupta@utdallas.edu

Abstract

Significant research has been conducted in recent years to extend Inductive Logic Programming (ILP) methods to induce a more expressive class of logic programs such as answer set programs. The methods proposed perform an exhaustive search for the correct hypothesis. Thus, they are sound but not scalable to real-life datasets. Lack of scalability and inability to deal with noisy data in real-life datasets restricts their applicability. In contrast, top-down ILP algorithms such as FOIL, can easily guide the search using heuristics and tolerate noise. They also scale up very well, due to the greedy nature of search for best hypothesis. However, in some cases despite having ample positive and negative examples, heuristics fail to direct the search in the correct direction. In this paper, we introduce the FOLD 2.0 algorithm—an enhanced version of our recently developed algorithm called FOLD. Our original FOLD algorithm automates the inductive learning of default theories. The enhancements presented here preserve the greedy nature of hypothesis search during clause specialization. These enhancements also avoid being stuck in local optima—a major pitfall of FOIL-like algorithms. Experiments that we report in this paper, suggest a significant improvement in terms of accuracy and expressiveness of the class of induced hypotheses. To the best of our knowledge, our FOLD 2.0 algorithm is the first heuristic based, scalable, and noise-resilient ILP system to induce answer set programs.

2012 ACM Subject Classification Computing methodologies → Inductive logic learning

Keywords and phrases Inductive Logic Programming, Negation As Failure, Answer Set Programming, Default reasoning, Machine learning

Digital Object Identifier 10.4230/OASIcs.ICLP.2018.2

Funding Authors are partially supported by NSF Grant IIS 171894

1 Introduction

Statistical machine learning methods produce models that 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. Lack of intuitive descriptions makes it hard for users to understand and verify the underlying rules that govern the model. Also, these methods cannot produce a justification for a prediction they compute for a new data sample. Additionally, extending prior knowledge (background knowledge) in these methods, requires the entire model to be relearned by adding new features to its *feature vector*. A feature vector is essentially *propositional* representation of data in statistical machine learning. In case of missing features, statistical methods such as Expectation Maximization (EM) algorithm are

Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018).
Editors: Alessandro Dal Palu', Paul Tarau, Neda Saeedloei, and Paul Fodor; Article No. 2; pp. 2:1–2:14
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

applied to fill the absent feature(s) with an average estimate that would maximize the likelihood of present features. This is fundamentally different from the human thought process that relies on common-sense reasoning. Humans generally do not directly perform probabilistic reasoning in the absence of information. Instead, most of the time human reasoning relies on learning default rules and exceptions.

Default Logic [15] is a *non-monotonic* logic to formalize reasoning with default assumptions. Normal logic programs provide a simple and practical formalism for expressing default rules. A default rule of the form $\frac{\alpha_1 \wedge \ldots \wedge \alpha_m : \neg\beta_{m+1}, \ldots, \neg\beta_n}{\gamma}$ can be formalized as the following normal logic program:

$$\gamma \leftarrow \alpha_1, ..., \alpha_m, not\ \beta_{m+1}, ..., not\ \beta_n$$

where γ, αs and βs are positive predicates.

Inductive Logic Programming (ILP) [9] is a sub-field of machine learning that mines data presented in the form of Horn clauses to learn hypotheses also as Horn clauses. However, Horn clause ILP is not expressive enough to induce default theories. Therefore, in order to learn default theories, an algorithm should be able to efficiently deal with *negation-as-failure* and normal logic programs [16].

Many researchers have tried to extend Horn ILP into richer non-monotonic logic formalisms. A survey of extending Horn clause based ILP to non-monotonic logics can be found in the work by Sakama [16]. He also proposes algorithms to learn from the answer set of a *categorical* normal logic program. He extends his algorithms in a framework called *brave* induction [17]. Law et. al. realized that this framework is not expressive enough to induce programs that solve practical problems such as combinatorial problems and proposed the ILASP system [4]. ASPAL [1] system is also an effort in this direction. Both ILASP and ASPAL encode the ILP instance as an ASP program and then they use an ASP solver to perform the exhaustive search of the correct hypothesis. This approach suffers from lack of scalability due to this exhaustive search. More discussion of advantages of our work presented in this paper *vis a vis* these earlier efforts is reported in Section 6.

The previous ILP systems are characterized as either bottom-up or top-down depending on the direction they guide the search. A bottom-up ILP system, such as Progol [10], builds most-specific clauses from the training examples. It is best suited for incremental learning from a few examples. In contrast, a top-down approach, such as the well-known FOIL algorithm [13], starts with the most-general clauses and then specializes them. It is better suited for large-scale datasets with noise, since the search is guided by heuristics [23].

In [20] we introduced an algorithm called FOLD that learns default theories in the form of stratified normal logic programs[1]. The default theories induced by FOLD, as well as the background knowledge used, is assumed to follow the stable model semantics [3]. FOLD extends the FOIL algorithm. FOLD can tolerate noise but it is not sound (i.e., there is no guarantee that the heuristic would always direct the search in the right direction). The *information gain* heuristic used in FOLD (that has been inherited from FOIL), has been extensively compared to other search heuristics in decision-tree induction [7]. There seems to be a general consensus that it is hard to improve the heuristic such that it would always select the correct literal to expand the current clause in specialization. The blame rests mainly on getting stuck in local optima, i.e, choosing a literal producing maximum information gain at a particular step that does not lead to a global optimum.

Similarly, in multi-relational datasets, a common case is that of a literal that has zero information gain but needs to be included in the learned theory. Heuristics-based algorithms will reject such a literal. Quinlan in [12] introduces *determinate literals* and suggests to add them all at once to the current clause to create a potential path towards a correct hypothesis. FOIL then requires a post

[1] Note that FOLD has been recently extended by us to learn arbitrary answer set programs, i.e., non-stratified ones too [19]; discussion of this extension is beyond the scope of this paper.

pruning phase to remove the unnecessary literals. This approach cannot trivially be extended to the case of default theories where determinate literals may appear in composite *abnormality* predicates and FOIL's language bias simply does not allow negated composite literals.

In this paper we present an algorithm called FOLD 2.0 which avoids being trapped in local optima and adds determinate literals while inducing default theories. We make the following novel contributions:

- We propose a new "cumulative" scoring function which replaces the original scoring function (called *information gain*). Our experiments show a significant improvement in terms of our algorithm's accuracy.
- We also extend FOLD with determinate literals. This extension enables FOLD to learn a broader class of hypotheses that, to the best of our knowledge, no other ILP system is able to induce. Finally, we apply our algorithm in variety of different domains including *kinship* and *legal* as well as UCI benchmark datasets to show how FOLD 2.0, significantly improves our algorithm's predictive power.

Rest of the paper is organized as follows: Section 2 presents background material. Section 3 introduces the FOLD algorithm. Section 4 presents the "cumulative" scoring function and determinate literals in FOLD 2.0. Section 5 presents our experiments and results. Section 6 discusses related research and Section 7 presents conclusions along with future research directions.

2 Background

Our original learning algorithm for inducing answer set programs, called FOLD (First Order Learning of Default rules) [20], is itself an extension of the well known FOIL algorithm. FOIL is a top-down ILP algorithm which 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). The FOIL algorithm learns a target predicate that has to be specified. Essentially, the target predicate appears as the head of the learned goal clause that FOIL aims to learn. A typical *stopping criterion* for the outer loop is determined as the coverage of all positive examples. Similarly, it can be specified as exclusion of all negative examples in the inner loop. The function $covers(\hat{c}, E^+, B)$ returns a set of examples in E^+ implied by the hypothesis $\hat{c} \cup B$.

The inner loop searches for a clause with the highest information gain using a general-to-specific hill-climbing search. To specialize a given clause c, a refinement operator ρ under θ-subsumption [11] is employed. The most general clause is {`p(`$X_1,...,X_n$`) :- true.`}, where the predicate `p/n` is the target and each X_i is a variable. The refinement operator specializes the current clause {`h :-` b_1`,...,`b_n`.`}. This is realized by adding a new literal `l` to the clause, which yields the following: {`h :-` b_1`,...,`b_n`,l`}. The heuristic based search uses *information gain*. In FOIL, information gain for a given clause is calculated as follows [8]:

$$IG(L,R) = t\left(log_2\frac{p_1}{p_1+n_1} - log_2\frac{p_0}{p_0+n_0}\right) \tag{1}$$

where L is the candidate literal to add to rule R, p_0 is the number of positive bindings of R, n_0 is the number of negative bindings of R, p_1 is the number of positive bindings of $R+L$, n_1 is the number of negative bindings of $R+L$, t is the number of positive bindings of 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:

Algorithm 1 Overview of the FOIL algorithm

Input: $goal, B, E^+, E^-$
Output: Hypothesis H

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

▶ Example 2.1. B, E^+ are background knowledge and positive examples respectively under *Closed World Assumption*, and the target predicate is `fly`.

```
B :   bird(X) :- penguin(X).   bird(tweety).     bird(et).
      cat(kitty).              penguin(polly).
E+ :  fly(tweety).             fly(et).
```

The FOIL algorithm would learn the following rule:

```
fly(X) :- not cat(X), not penguin(X).
```

which does not yield a constructive definition. The best theory in this example is as follows:

```
fly(X):- bird(X), not penguin(X).
```

which FOIL fails to discover.

3 FOLD Algorithm

The intuition behind FOLD algorithm is to learn a concept in terms of a default and possibly multiple exceptions (and exceptions to exceptions, and so on). Thus, in the bird example given above, we would like to learn the rule that `X` flies if it is a bird and not a penguin, rather than that all non-cats and non-penguins can fly. FOLD tries first to learn the default by specializing a general rule of the form `{goal(`$V_1, ..., V_n$`) :- true.}` with positive literals. As in FOIL, each specialization must rule out some already covered negative examples without significantly decreasing the number of positive examples covered. Unlike FOIL, no negative literal is used at this stage. Once the IG becomes zero, this process stops. At this point, if any negative example is still covered, they must be either noisy data or exceptions to the current hypothesis. Exceptions are separated from noise via distinguishable patterns in negative examples [21]. In other words, exceptions can be learned by swapping of positive and negative examples and calling the same algorithm recursively. This swapping of positive and negative examples and then recursively calling the algorithm again can continue, so that we can learn exceptions to exceptions, and so on. Each time a rule is discovered for exceptions, a new predicate `ab(`$V_1, ..., V_n$`)` is introduced. To avoid name collisions, FOLD appends a unique number at the end of the string "ab" to guarantee the uniqueness of invented predicates. It turns out that the outlier data samples are covered neither as default nor as exceptions. If outliers are present, FOLD identifies and enumerates them to make sure that the algorithm converges. This ability to separate exceptions from noise allows FOLD (and FOLD 2.0, introduced later) pinpoint noise more accurately. This is in contrast to FOIL, where exceptions and noisy data are clubbed together. Details can be found in [20].

Algorithm 2 shows a high level implementation of the FOLD algorithm. In lines 1-8, function FOLD, serves like the FOIL outer loop. In line 3, FOLD starts with the most general clause (e.g. `fly(X) :- true`). In line 4, this clause is refined by calling the function *SPECIALIZE*. In lines 5-6, 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 like 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 $first\_iteration$ 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 E^+ and E^-.

In line 31, the "best" positive literal that covers more positive examples and fewer negative examples is selected. Again, note 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 at lines 35-36 it is associated with the body of the rule(s) found by the recurring FOLD function call at line 33. Finally, at line 38, default and exception are combined together to form a single clause.

Now, we illustrate how FOLD discovers the above set of clauses given $E^+ = \{tweety, et\}$ and $E^- = \{polly, kitty\}$ and the goal `fly(X)`. By calling FOLD, at line 2 while loop, the clause `{fly(X) :- true.}` is specialized. Inside the *SPECIALIZE* function, at line 12, the literal `bird(X)` is selected to add to the current clause, to get the clause $\hat{c}$ = `fly(X) :- bird(X)`, which happens to have the greatest IG among `{bird,penguin,cat}`. Then, at lines 26-27 the following updates are performed: $E^+ = \{\}$, $E^- = \{polly\}$. 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 E^+, E^-. Now, FOLD is recursively called to learn a rule for $E^+ = \{polly\}$, $E^- = \{\}$. The recursive call (line 33), returns `{fly(X) :- penguin(X)}` as the exception. In line 34, a new predicate `ab0` is introduced and at lines 35-37 the clause `{ab0(X) :- 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 following theory:

190
```
fly(X) :- bird(X), not ab0(X).
ab0(X) :- penguin(X).
```

More detailed examples can be found in [20].

4 The FOLD 2.0 Algorithm

4.1 Cumulative Scoring Function

The *kinship* domain is one of the initial successful applications of the FOIL algorithm [13], where the algorithm learns general rules governing social interactions and relations (particularly kinship) from a series of examples. For example, it can learn the "Uncle" relationship, given the background

Algorithm 2 FOLD Algorithm

Input: $goal, B, E^+, E^-$

Output:

$D = \{c_1, ..., c_n\}$ ▷ defaults' clauses

$AB = \{ab_1, ..., ab_m\}$ ▷ exceptions/abnormal clauses

function FOLD(E^+, E^-)
  while ($size(E^+) > 0$) **do**
    $c \leftarrow (goal$:- $true.)$
    $\hat{c} \leftarrow$ SPECIALIZE(c, E^+, E^-)
    $E^+ \leftarrow E^+ \setminus covers(\hat{c}, E^+, B)$
    $D \leftarrow D \cup \{\hat{c}\}$
  end while
end function
function SPECIALIZE(c, E^+, E^-)
  $first\_iteration \leftarrow true$
  while ($size(E^-) > 0$) **do**
    $(c_{def}, \hat{IG}) \leftarrow$ ADD_BEST_LITERAL(c, E^+, E^-)
    if $\hat{IG} > 0$ **then**
      $\hat{c} \leftarrow c_{def}$
    else
      if $first\_iteration$ **then**
        $\hat{c} \leftarrow enumerate(c, E^+)$
      else
        $\hat{c} \leftarrow$ EXCEPTION(c, E^-, E^+)
        if $\hat{c} = null$ **then**
          $\hat{c} \leftarrow enumerate(c, E^+)$
        end if
      end if
    end if
    $first\_iteration \leftarrow false$
    $E^+ \leftarrow E^+ \setminus covers(\hat{c}, E^+, B)$
    $E^- \leftarrow covers(\hat{c}, E^-, B)$
  end while
end function
function EXCEPTION(c_{def}, E^+, E^-)
  $\hat{IG} \leftarrow$ ADD_BEST_LITERAL(c, E^+, E^-)
  if $\hat{IG} > 0$ **then**
    $c\_set \leftarrow$ FOLD(E^+, E^-)
    $c\_ab \leftarrow generate\_next\_ab\_predicate()$
    for each $c \in c\_set$ **do**
      $AB \leftarrow AB \cup \{c\_ab$:- $bodyof(c)\}$
    end for
    $\hat{c} \leftarrow (headof(c_{def})$:- $bodyof(c)$, **not**$(c\_ab))$
  else
    $\hat{c} \leftarrow null$
  end if
end function

knowledge of “Brother", “Sister", “Father", “Mother", “Husband", “Wife" and some positive and negative examples of the concept. However, if the background knowledge only contains the primitive relationships including “Sibling", “Parent", “Married" and gender descriptors, it fails to discover the correct rule for “Uncle". As an experiment, we used an arbitrarily produced kinship dataset only containing the primitive relationships. The FOIL algorithm produced the following rules:

Rule (1) `uncle(A,B) :- male(A), parent(A,_), female(B).`
Rule (2) `uncle(A,_) :- male(A), parent(A,B), female(B), sibling(B,_).`

Similarly, the FOLD algorithm found incorrect rules as follows:

Rule (1) `uncle(V1,V2) :- male(V1), parent(V2,V3).`
Rule (2) `uncle(V1,V2) :- male(V1), parent(V2,V3), female(V2).`

Table 1 shows the *information gain* for each candidate literal while discovering Rule (1). At first iteration, the algorithm successfully finds the literal `male(V1)`, because it has the maximum gain

Literal / Clause	uncle(V1,V2) :- true	uncle(V1,V2) :- male(V1)
parent(V1,V3)	1.44	1.01
parent(V2,V3)	1.06	**1.16**
parent(V3,V1)	1.44	1.01
sibling(V1,V3)	2.27	1.01
sibling(V3,V1)	2.27	1.01
male(V1)	**3.18**	-
female(V2)	0.34	0.50
married(V1,V3)	0.69	0
married(V2,V3)	0.34	0.50
married(V3,V1)	0.69	0
married(V3,V2)	0.34	0.5

Table 1 FOLD Execution to Discover Rule (1)

($IG = 3.18$). At second iteration, the literal `parent(V2,V3)` has the highest gain ($IG = 1.16$) and hence is selected. At this point, since the rule does not cover any negative example, the algorithm returns. This example characterizes a case in which the highest score does not correspond to the correct literal. The correct literal at second iteration is `sibling(V1,V3)`, whose information gain is 1.01 and it is less than the maximum.

We observed that neither increasing the number of examples nor changing the scoring function would solve this problem. As an experiment, we replaced the *information gain* with other scoring functions reported in the literature including *Matthews Correlation Coefficient* (MCC), F_β-measure [23] and the FOSSIL [2] scoring measure based on statistical correlation. They all suffer from the same problem.

A key observation is the following: as more literals are introduced, the number of positive and negative examples covered by the current clause shrinks. With fewer examples, the accuracy of heuristic decreases too. In Table 1, `sibling(V1,V3)` should have had the highest score at second iteration. At first iteration, `sibling(V1,V3)` ranks second after `male(V1)`. A simple comparison between the score of `sibling(V1,V3)` and `parent(V2,V3)` shows the former provides better coverage (exclusion) of positive (negative) examples than the latter. But the algorithm is oblivious of this information at the beginning of second iteration as it goes only by magnitude of the scoring function for the current iteration. This score becomes less and less accurate as more literals are introduced and fewer examples remain to cover. If the algorithm could remember that at first iteration, `sibling(V1,V3)` was able to cover/exclude the examples much better than `parent(V2,V3)`, it would prefer `sibling(V1,V3)` over `parent(V2,V3)`.

To concretize this, we propose the idea of keeping a *cumulative score*, i.e., to transfer a portion of past score (if one exists) to the value that the scoring function computes for current iteration. Our experiments suggest that there is not a universal optimal value that would always result in highest accuracy. In other words, the optimal value varies from a dataset to another. Thus, in order to implement the “cumulative score", we introduce a new hyperparameter[2], namely, α, whose value is decided via cross-validation of the dataset being used. In order to compute the score of each literal during the search, the *information gain* is replaced with “cumulative gain".

[2] In Machine Learning, a hyperparameter is a parameter whose value is set before the learning process begins.

Literal / Clause	uncle(V1,V2).	uncle(V1,V2):- male(V1)	uncle(V1,V2):-male(V1), sibling(V1,V3)
parent(V1,V3)	1.44	1.30	0
parent(V2,V3)	1.06	1.38	0
parent(V3,V2)	0	0	**2.49**
parent(V3,V1)	1.44	1.30	0
parent(V2,V4)	-	-	0.83
sibling(V1,V3)	2.27	**1.47**	-
sibling(V3,V1)	2.27	1.47	1.15
male(V1)	**3.18**	-	-
female(V2)	0.34	0.57	0
female(V3)	-	-	1.15
married(V1,V3)	0.69	0	0
married(V2,V3)	0.34	0.57	0
married(V3,V1)	0.69	0	0
married(V3,V2)	0.34	0.57	0
married(V2,V4)	-	-	1.24
married(V4,V2)	-	-	1.24

Table 2 FOLD 2.0 Execution with Cumulative Score

Formally, let R_i denote the induced rule up until iteration $i+1$ of FOLD's inner loop execution. Thus, R_0 is the rule `{goal :- true.}`. Also, let $score_i(R_{i-1},L)$ denote the score of literal L in clause R_{i-1} at iteration i of FOLD's inner loop execution. The "cumulative" score at iteration $i+1$ for literal l is computed as follows:

$$score_{i+1}(R_i,L) = IG(R_i,L) + \alpha \times score_i(R_{i-1},L)$$

If $score_i(R_{i-1},L)$ does not exist, it is considered as zero. Also, if $IG(R_i,L) = 0$, the "cumulative" score from the past is not taken into account. Initially, the cumulative score is considered zero for all candidate literals. Table 2 shows the FOLD 2.0 algorithm's execution to learn "uncle" predicate on the same dataset. With choice of $\alpha = 0.2$, the algorithm is able to discover the following rule: `uncle(V1,V2) :- male(V1), sibling(V1,V3), parent(V3,V2).` It should also be noted that only promising literals are shown in Table 1 and 2. Next, we discuss how our FOLD 2.0 algorithm handles zero information-gain literals.

4.2 Extending FOLD with Determinate Literals

A literal in the body of a clause can serve two purposes: (i) it may contribute directly to the inclusion/exclusion of positive/negative examples respectively; or, (ii) it may contribute indirectly by introducing new variables that are used in the subsequent literals. This type of literal may or may not yield a positive score. Therefore, it is quite likely that our hill-climbing algorithm would miss them. Two main approaches have been used to take this issue into account: *determinate literals* [12] and *lookahead* technique [6]. The latter technique is not of interest to us because it does not preserve the greedy nature of search.

Determinate literals are of the form `r(X,Y)`, where `r/2` is a new literal introduced in the hypothesis' body and Y is a new variable. The literal `r/2` is determinate if, for every value of `X`, there is at most one value for `Y`, when the hypothesis' head is unified with positive examples. Determinate literals are not contributing directly to the learning process, but they are needed as they influence the literals chosen in the future. Since their inclusion in the hypothesis is computationally inexpensive, the FOIL algorithm adds them to the hypothesis simultaneously. In Section 2 we showed why the naive handling of negation in FOIL would not work in case of non-monotonic logic programs. Another issue with FOIL's handling of negated literals arises when we deal with *determinate literals*. Whenever a combination of a determinate and a gainful literal attempts to find a pattern in the negative examples, the FOIL algorithm fails to discover it because FOIL prohibits conjunction of negations in its language bias to prevent search space explosion. However, by introducing the abnormality predicates and recursively swapping positive and negative examples, FOLD makes inductive learning of such default theories possible.

The FOLD algorithm always selects literals with positive information gain first. Next, if some negative examples are still covered and no gainful literal exists, it would swap the current positive examples with current negative examples and recursively calls itself to learn the exceptions. To accommodate determinate literals in FOLD 2.0, we make the following modification to FOLD. In the SPECIALIZE function, right before swapping the examples and making the recursive call to the FOLD function (see Algorithm 3), we try the current rule for a second time. By adding determinate literals and iterating again, we hope that a positive gainful literal will be discovered. Next, if that choice does not exclude the negative examples, FOLD 2.0 swaps the examples and recursively calls itself. A nice property of this recursive approach is that the determinate literals might be added inside the exception finding routine to induce a composite abnormality predicate. Neither FOIL nor FOLD could induce such hypotheses. The following example shows how this is handled in the FOLD 2.0 algorithm.

▶ Example 4.1. In United States immigration system, student visa holders are classified as F1(student) and F2(student's spouse). F1 and F2 status remains valid until a student graduates. The spouse of such an individual maintains a valid status, as long as that individual is a student. Table 3 shows a dataset for this domain. In this dataset, it turns out that $married(V1,V2)$ is a determinate literal and essential to the final hypothesis. If we run the FOLD 2.0 algorithm, it would produce the following hypothesis:

286

Default rule(1):	`valid(V1) :- student(V1), not ab1(V1).`
Default rule(2):	`valid(V1) :- class(V1,f2), not ab2(V1).`
Exception(1) :	`ab1(V1) :- graduated(V1).`
Exception(2) :	`ab2(V1) :- married(V1,V2), graduated(V2).`

In this example default rule(1) as well as rules for its exception are discovered first. This rule (rule(1)) takes care of students who have not graduated yet. Then, while discovering rule(2), after choosing the only gainful literal, i.e., `class(V1,f2)`, the algorithm is recursively called on the exception part. It turns out that there is no gainful literal that covers the now positive examples (previously negative examples). The only determinate literal in this example is `married(V1,V2)`, which is added at this point. This is followed by FOLD 2.0 finding a gainful literal, i.e., `graduated(V2)`, and then returning the default rule(2). At this point, all positive examples are covered and the algorithm terminates. Default rule(2) takes care of the class of F2 visa holders whose spouse is a student unless they have graduated. The Algorithm 3 shows the changes necessary to the FOLD algorithm in order to handle determinate literals.

5 Experiments and results

In this section we present our experiments on UCI benchmark datasets [5]. Table 4 summarizes an accuracy-based comparison between Aleph [21], FOLD [20] and FOLD 2.0. We report a significant improvement just by picking up an optimal value for α via cross-validation. In these experiments we picked $\alpha \in \{0, 0.2, 0.5, 0.8, 1\}$.

Algorithm 3 Overview of FOLD 2.0 Algorithm + determinate literals

Input: $goal, B, E^+, E^-$

Output: $D = \{c_1, ..., c_n\}$, $AB = \{ab_1, ..., ab_m\}$

```
function SPECIALIZE(c, E+, E-)
    determinate_added ← false
    while (size(E-) > 0) do
        (c_def, IĜ) ← ADD_BEST_LITERAL(c, E+, E-)
        if IĜ ≤ 0 then
            if determinate_added == false then
                c ← ADD_DETERMINATE_LITERALS(c, E+, E-)
                determinate_added ← true
            else
                ĉ ← EXCEPTION(c, E-, E+)
                if ĉ = null then
                    ĉ ← enumerate(c, E+)
                end if
            end if
        else
            E+ ← E+ \ covers(ĉ, E+, B)
            E- ← covers(ĉ, E-, B)
        end if
    end while
end function
```

ILP algorithms usually achieve lower accuracy compared to state-of-the-art statistical methods such as SVM. But in case of "Post Operative" dataset, for instance, our FOLD 2.0 algorithm outperforms SVM, whose accuracy is only 67% [18]. Next, we show in detail how FOLD 2.0 achieves higher accuracy in case of Moral Reasoner dataset. Moral Reasoner is a rule-based model that qualitatively simulates moral reasoning. The model was intended to simulate how an ordinary person, down to about age five, reasons about harm-doing. The Horn-clause theory has been provided along with 202 instances that were used in [22]. The top-level predicate to predict is `guilty/1`. We encourage the interested reader to refer to [5] for more details. Our goal is to learn the moral reasoning behavior from examples and check how close it is to the Horn-clause theory reported in [22].

B				E^+	E^-
class(p1,f2).	class(p7,f1).	student(p3).	married(p1,p2).	valid(p1).	valid(p4).
class(p2,f1).	class(p8,f1).	student(p4).	married(p5,p6).	valid(p2).	valid(p5).
class(p3,f1).	class(p9,f2).	student(p6).	married(p9,p10).	valid(p3).	valid(p6).
class(p4,f1).	class(p10,f1).	student(p7).	graduated(p4).	valid(p7).	valid(p8).
class(p5,f2).		student(p8).	graduated(p6).	valid(p9).	
class(p6,f1).		student(p10).	graduated(p8).	valid(p10).	

Table 3 Valid Student Visa Dataset

First, we run FOLD 2.0 algorithm with $\alpha = 0$. This literally turns off the "cumulative score" feature. The algorithm would return the following set of rules:

```
Rule(1) guilty(V1) :- severity(V1,1), external_force(V1,n),
                      benefit_victim(V1,0),intervening_contribution(V1,n).
Rule(2) guilty(V1) :- severity(V1,1), external_force(V1,n),
                      benefit_victim(V1,0),foresee_intervention(V1,y).
Rule(3) guilty(V1) :- someone_else_cause_harm(V1,y),achieve_goal(V1,n),
                      control_perpetrator(V1,y), foresee_intervention(V1,n).
```

In the original Horn clause theory [22] there are two theories for being guilty: i) blameworthy, ii) vicarious_blame. The following rules for `blame_worthy(X)` are reproduced from [22]:

```
blameworthy(X):- responsible(X), not justified(X), severity_harm(X,H),
                 benefit_victim(X,L), H > L.
responsible(X):- cause(X), not accident(X), external_force(X,n),
                 not intervening_cause(X).
intervening_cause(X) :- intervening_contribution(X,y),
                        forsee_intervention(X).
```

Rule(1) and Rule(2), that FOLD 2.0 learns, together build the blameworthy definition of the original theory. The predicates `severity_harm` and `benefit_victim` occur in Rule(1) and Rule(2). It should be noted that due to the nature of the provided examples, FOLD 2.0 comes up with a more specific version compared to the original theory reported in [22]. In addition, instead of learning the predicate `responsible(X)`, our algorithm learns its body literals. The predicate `cause(X)` does not appear in the hypothesis because it is implied by all positive and negative examples, one way or another. The predicate `not intervening_cause(X)` appears in our hypothesis due to application of *De Morgan's law* and flipping yes and no in the second arguments. The rest of the guilty cases fall into the category of `vicarious_blame` below:

```
vicarious_blame(X):- vicarious(X),
        not justified(X),
        severity_harm(X,H),
        benefit_victim(X,L), H > L.

vicarious(X) :-
        someone_else_cause_harm(X,y),
        outrank_perpetrator(X,y),
        control_perpetrator(X,y).
```

There is a discrepancy in Rule(3), compared to the corresponding `vicarious_blame` in the original theory. However, by setting the cumulative score parameter $\alpha = 0.2$, FOLD 2.0 would produce the following set of rules:

```
Rule(1):
guilty(V1) :- severity_harm(V1,1),
    external_force(V1,n),
    benefit_victim(V1,0),
    intervening_contribution(V1,n).

Rule(2):
   guilty(V1) :-
   severity_harm(V1,1),
   external_force(V1,n),
   benefit_victim(V1,0),
   foresee_intervention(V1,y).

Rule(3):
guilty(V1) :- severity_harm(V1,1), benefit_victim(V1,0),
              someone_else_cause_harm(V1,y),outrank_perpetrator(V1,y),
              control_perpetrator(V1,y).
```

Rule(1) and Rule(2) are generated in FOLD 2.0 as before. However, Rule(3) perfectly matches that of the original theory which our FOLD algorithm would have not been able to discover without

Dataset	Accuracy (%)			α
	Aleph	FOLD	FOLD 2.0	
Labor	85	94	100	0.5
Post-op	62	65	78	1
Bridges	89	90	93	1
Credit-g	70	78	84	0.5
Moral	96	96	100	0.2

Table 4 Performance Results on UCI Benchmark Datasets

"cumulative score". Note that the cumulative score heuristics is quite general and can be used to enhance any machine learning algorithm that relies on the concept of information gain. In particular, it can be used to improve the FOIL algorithm itself.

6 Related Work

A survey of non-monotonic ILP work can be found in [16]. Sakama also introduces an algorithm to induce rules from answer sets. His approach may yield premature generalizations that include redundant negative literals. We skip the illustrative example due to lack of space, however, the reader can refer to [20]. ASPAL [1] is another ILP system capable of producing non-monotonic logic programs. It encodes ILP problem as an ASP program. XHAIL [14] is another ILP system that heavily uses abductive logic programming to search for the best hypothesis. Both ASPAL and XHAIL systems can only learn hypotheses that have a single stable model. ILASP [4] is the successor of ASPAL. It can learn hypotheses that have multiple stable models by employing brave induction [17]. All of these systems perform an exhaustive search to find the correct hypothesis. Therefore, they are not scalable to real-life datasets. They also have a restricted language bias to avoid the explosion of search space of hypotheses. This overly restricted language bias does not allow them to learn new predicates, thus keeping them from inducing sophisticated default theories with nested or composite abnormalities that our FOLD 2.0 algorithm can induce. For instance consider the following example, a default theory with abnormality predicate represented as conjunction of two other predicates, namely `s(X) and r(X)`.

375
```
p(X) :- q(X), not ab(X).
ab(X) :- s(X), r(X).
```

Our algorithm has advantages over the above mentioned systems: It follows a greedy top-down approach and therefore it is better suited for larger datasets and noisy data. Also, it can invent new predicates [19], distinguish noise from exceptions, and learn nested levels of exceptions.

7 Conclusion and Future Work

In this paper we presented *cumulative score*-based heuristic to guide the search for best hypothesis in a top-down non-monotonic ILP setting. The main feature of this heuristic is that it avoids being trapped in local optima during clause specialization search. This results in significant improvement in the accuracy of induced hypotheses. This heuristic is quite general and can be used to enhance any machine learning algorithm that relies on the concept of information gain. In particular, it can be used to improve the FOIL algorithm itself. We used it in this paper to extend our FOLD algorithm to obtain the FOLD 2.0 algorithm for learning answer set programs. FOLD 2.0 performs significantly better than our FOLD algorithm [20], where the FOLD algorithm itself produces better results than previous systems such as FOIL and ALEPH. We also showed how determinate literals can be adapted to identifying patterns in negative examples after the swapping of positive and negative examples in FOLD. Note that while determinate literals were introduced in the FOIL algorithm, their use in FOIL was limited to only positive literals. Generalizing the use of determinate literals in FOLD 2.0, enables us to induce hypotheses that no other non-monotonic ILP system is able to induce.

There are three main avenues for future work: (i) handling large datasets using methods similar to QuickFoil [23]. In QuickFoil, all the operations of FOIL are performed in a database engine. Such an implementation, along with pruning techniques and query optimization tricks, can make the FOLD 2.0 training phase much faster. (ii) FOLD 2.0 learns function-free answer set programs. We plan to investigate extending the language bias towards accommodating functions. (iii) Combining statistical methods such as SVM with FOLD 2.0 to increase accuracy as well as providing explanation for the behavior of models produced by SVM.

References

1. Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive logic programming in answer set programming. In Stephen Muggleton, Alireza Tamaddoni-Nezhad, and Francesca A. Lisi, editors, *Inductive Logic Programming - 21st International Conference, ILP 2011, Windsor Great Park, UK, July 31 - August 3, 2011, Revised Selected Papers*, volume 7207 of *Lecture Notes in Computer Science*, pages 91–97. Springer, 2011. URL: `https://doi.org/10.1007/978-3-642-31951-8_12`, `doi:10.1007/978-3-642-31951-8_12`.
2. Johannes Fürnkranz. Fossil: A robust relational learner. In Francesco Bergadano and Luc De Raedt, editors, *Machine Learning: ECML-94*, pages 122–137, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
3. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.
4. Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, volume 8761 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2014. URL: `https://doi.org/10.1007/978-3-319-11558-0_22`, `doi:10.1007/978-3-319-11558-0_22`.
5. M. Lichman. UCI machine learning repository, 2013. URL: `http://archive.ics.uci.edu/ml`.
6. Marco Lippi, Manfred Jaeger, Paolo Frasconi, and Andrea Passerini. Relational information gain. *Machine Learning*, 83(2):219–239, May 2011. URL: `https://doi.org/10.1007/s10994-010-5194-7`, `doi:10.1007/s10994-010-5194-7`.
7. John Mingers. An empirical comparison of selection measures for decision-tree induction. *Machine Learning*, 3:319–342, 1989.
8. Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.
9. Stephen Muggleton. Inductive logic programming. *New Generation Comput.*, 8(4):295–318, 1991. URL: `http://dx.doi.org/10.1007/BF03037089`, `doi:10.1007/BF03037089`.
10. Stephen Muggleton. Inverse entailment and progol. *New Generation Comput.*, 13(3&4):245–286, 1995. URL: `http://dx.doi.org/10.1007/BF03037227`, `doi:10.1007/BF03037227`.
11. G. D. Plotkin. A further note on inductive generalization, in machine intelligence, volume 6, pages 101-124, 1971.
12. J. R. Quinlan. Determinate literals in inductive logic programming. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'91, pages 746–750, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. URL: `http://dl.acm.org/citation.cfm?id=1631552.1631572`.

13 J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990. URL: `http://dx.doi.org/10.1007/BF00117105`, `doi:10.1007/BF00117105`.

14 Oliver Ray. Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, 7(3):329 – 340, 2009. Special Issue: Abduction and Induction in Artificial Intelligence. URL: `http://www.sciencedirect.com/science/article/pii/S1570868308000682`, `doi:http://dx.doi.org/10.1016/j.jal.2008.10.007`.

15 Raymond Reiter. A logic for default reasoning. *Artif. Intell.*, 13(1-2):81–132, 1980. URL: `https://doi.org/10.1016/0004-3702(80)90014-4`, `doi:10.1016/0004-3702(80)90014-4`.

16 Chiaki Sakama. Induction from answer sets in nonmonotonic logic programs. *ACM Trans. Comput. Log.*, 6(2):203–231, 2005. URL: `http://doi.acm.org/10.1145/1055686.1055687`, `doi:10.1145/1055686.1055687`.

17 Chiaki Sakama and Katsumi Inoue. Brave induction: a logical framework for learning from incomplete information. *Machine Learning*, 76(1):3–35, 2009. URL: `https://doi.org/10.1007/s10994-009-5113-y`, `doi:10.1007/s10994-009-5113-y`.

18 Mathieu Serrurier and Henri Prade. Introducing possibilistic logic in ILP for dealing with exceptions. *Artificial Intelligence*, 171:939–950, 2007.

19 Farhad Shakerin and Gopal Gupta. Technical report, heuristic based induction of answer set programs: From default theories to combinatorial problems, http://arxiv.org/abs/1802.06462, 2018. URL: `http://arxiv.org/abs/1802.06462`.

20 Farhad Shakerin, Elmer Salazar, and Gopal Gupta. A new algorithm to automate inductive learning of default theories. *TPLP*, 17(5-6):1010–1026, 2017. URL: `https://doi.org/10.1017/S1471068417000333`, `doi:10.1017/S1471068417000333`.

21 Ashwin Srinivasan, Stephen Muggleton, and Michael Bain. Distinguishing exceptions from noise in non-monotonic learning,in s. muggleton and k. furukawa, editors, second international inductive logic programming workshop (ilp92), 1996.

22 James Wogulis. Revising relational domain theories. In Lawrence Birnbaum and Gregg Collins, editors, *Proceedings of the Eighth International Workshop (ML91), Northwestern University, Evanston, Illinois, USA*, pages 462–466. Morgan Kaufmann, 1991.

23 Qiang Zeng, Jignesh M. Patel, and David Page. Quickfoil: Scalable inductive logic programming. *Proc. VLDB Endow.*, 8(3):197–208, November 2014. URL: `http://dx.doi.org/10.14778/2735508.2735510`, `doi:10.14778/2735508.2735510`.