FOLD-R++: A Toolset for Automated Inductive Learning of Default Theories from Mixed Data

Abstract

FOLD-R is an automated inductive learning algorithm for learning default rules with exceptions for mixed (numerical and categorical) data. It generates an (explainable) answer set programming (ASP) rule set for classification tasks. We present an improved FOLD-R algorithm, called FOLD-R++, that significantly increases the efficiency and scalability of FOLD-R. FOLD-R++ improves upon FOLD-R without compromising or losing information in the input training data during the encoding or feature selection phase. The FOLD-R++ algorithm is competitive in performance with the widely-used XGBoost algorithm, however, unlike XGBoost, the FOLD-R++ algorithm produces an explainable model. Next, we create a powerful tool-set by combining FOLD-R++ with s(CASP)—a goal-directed ASP execution engine—to make predictions on new data samples using the answer set program generated by FOLD-R++. The s(CASP) system also produces a justification for the prediction. Experiments presented in this paper show that our improved FOLD-R++ algorithm is a significant improvement over the original design and that the s(CASP) system can make predictions in an efficient manner as well.

1 Introduction

Dramatic success of machine learning has led to a torrent of Artificial Intelligence (AI) applications. However, the effectiveness of these systems is limited by the machines' current inability to explain their decisions and actions to human users. That's mainly because the statistical machine learning methods produce models that are complex algebraic solutions to optimization problems such as risk minimization or geometric margin maximization. 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 arrive at for a new data sample.

The Explainable AI program (Gunning 2015) aims to create a suite of machine learning techniques that: a) Produce more explainable models, while maintaining a high level of prediction accuracy. b) Enable human users to understand, appropriately trust, and effectively manage the emerging generation of artificially intelligent partners. Inductive Logic Programming (ILP) (Muggleton 1991) is one Machine Learning technique where the learned model is in the form of logic programming rules (Horn Clauses) that are comprehensible to humans. It allows the background knowledge to be incrementally extended without requiring the entire model to be re-learned. Meanwhile, the comprehensibility of symbolic rules makes it easier for users to understand and verify induced models and refine them.

The ILP learning problem can be regarded as a search problem for a set of clauses that deduce the training examples. The search is performed either top down or bottom-up. A bottom-up approach builds most-specific clauses from the training examples and searches the hypothesis space by using generalization. This approach is not applicable to large-scale datasets, nor it can incorporate *negation-as-failure* into the hypotheses. A survey of bottom-up ILP systems and their shortcomings can be found at (Sakama 2005). In contrast, top-down approach starts with the most general clause and then specializes it. A top-down algorithm guided by heuristics is better suited for large-scale and/or noisy datasets (Zeng, Patel, and Page 2014).

The FOIL algorithm (Quinlan 1990) by Quinlan is a popular top-down inductive logic programming algorithm that generate logic programs. FOIL uses weighted information gain as the heuristics to guide the search for best literals. The FOLD algorithm by Shakerin (Shakerin 2020; Shakerin, Salazar, and Gupta 2017) is a new top-down algorithm inspired by the FOIL algorithm. It generalizes the FOIL algorithm by learning default rules with exceptions. It does so by first learning the default conclusion that covers positive examples while avoiding negative examples, then next it swaps the positive and negative examples and calls itself recursively to learn the exceptions to the default conclusions. Both FOIL and FOLD cannot deal with numeric features directly; an encoding process is needed in the preparation phase of the training data that discretizes the continuous numbers into intervals. However, this process not only adds a huge computational overhead to the algorithm but also leads to loss of information in the training data.

To deal with the above problems, Shakerin developed an extension of the FOLD algorithm, called FOLD-R, to handle mixed (i.e., both numerical and categorical) features which avoids the discretization process for numerical data (Shakerin 2020; Shakerin, Salazar, and Gupta 2017). However, FOLD-R still suffers from efficiency and scalability issues when compared to other popular machine learning systems for classification. In this paper we report on a novel imple-

mentation method we have developed to improve the design of the FOLD-R system. In particular, we use the prefix sum technique (Wikipedia contributors 2021) to optimize the process of calculation of information gain, the most time consuming component of the FOLD family of algorithms (Shakerin 2020). Our optimization, in fact, reduces the time complexity of the algorithm. If `N` is the number of unique values from a specific feature and `M` is the number of training examples, then the complexity of classifying all the examples for each split is reduced from `O(M*N)` for FOLD-R to `O(M+N)` in FOLD-R++.

Our experimental results indicate that the FOLD-R++ algorithm is comparable to popular machine learning algorithms such as XGBoost wrt various metrics (accuracy, recall, precision, and F1-score) as well as in efficiency and scalability. However, in addition, FOLD-R++ produces an explainable and interpretable model in the form of an answer set program.

This paper makes the following novel contribution: it presents the FOLD-R++ algorithm that significantly improves the efficiency and scalability of the FOLD-R ILP algorithm without adding overhead during pre-processing or losing information in the training data. As mentioned, the new approach is competitive with popular classification models such as the XGBoost classifier (Chen and Guestrin 2016). The FOLD-R++ algorithm outputs an answer set program (ASP) (Gelfond and Kahl 2014) that serves as an explainable/interpretable model. This generated answer set program is compatible with s(CASP) (Arias et al. 2018), a goal-directed ASP solver, that can efficiently justify the prediction generated by the ASP model.[1]

2 Inductive Logic Programming

Inductive Logic Programming (ILP) (Muggleton 1991) is a subfield of machine learning that learns models in the form of logic programming rules (Horn Clauses) that are comprehensible to humans. This problem is formally defined as:
Given

1. A background theory B, in the form of an extended logic program, i.e., clauses of the form $h \leftarrow l_1, ..., l_m,\ not\ l_{m+1}, ...,\ not\ l_n$, where $l_1, ..., l_n$ are positive literals and *not* denotes *negation-as-failure* (NAF) (Baral 2003; Gelfond and Kahl 2014). We require that B has no loops through negation, i.e., it is stratified.
2. Two disjoint sets of ground target predicates E^+, E^- known as positive and negative examples, respectively
3. A hypothesis language of function free predicates L, and a refinement operator ρ under $\theta - subsumption$ (Plotkin 1971) that would disallow loops over negation.

Find a set of clauses H such that:

- $\forall e \in\ E^+,\ B \cup H \models e$
- $\forall e \in\ E^-,\ B \cup H \not\models e$
- $B \wedge H$ is consistent.

[1]The s(CASP) system is freely available at `https://gitlab.software.imdea.org/ciao-lang/sCASP`.

3 The FOLD-R++ Algorithm

The FOLD algorithm (Shakerin 2020; Shakerin, Salazar, and Gupta 2017) is a top-down ILP algorithm that searches for best literals to add to the body of the clauses for hypothesis, H, with the guidance of an information gain-based heuristic. The FOLD-R++ algorithm[2] refactors the FOLD algorithm and is summarized in algorithm1. The output of the FOLD-R++ algorithm is a set of default rules that include exceptions. An example implied by any rule in the set would be classified as positive. Therefore, the FOLD-R++ algorithm rules out the already covered positive examples in line 5 after learning a new rule. For each rule learning process, a best literal would be selected based on weighted information gain with the current training examples, in line 13, then the examples that cannot be implied by learned default literals would be ruled out for further learning of the current rule. When the information gain becomes zero or the number of negative examples drops below the ratio threshold, the default learning part is done. Unlike the FOIL algorithm, FOLD-R++ next learns exceptions after first learning default literals. This is done by swapping the residual positive and negative examples and calling itself recursively in line 29. The remaining positive and negative examples can be swapped again and exceptions to exceptions learned (and then swapped further to learn exceptions to exceptions of exceptions, and so on). The $ratio$ parameter in Algorithm 1 represents the ratio of training examples that are part of the exception to the examples implied by only the default conclusion part of the rule. It will allow us to control the nesting level of exceptions the user wants to permit.

Example 1 *In the FOLD-R++ algorithm, the target is to learn rules for* `fly(X)`*.* B, E^+, E^- *are background knowledge, positive and negative examples, respectively.*

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

The target predicate {`fly(X) :- true.`} is specified when calling the learn_rule function at line 4. The function selects the literal `bird(X)` as result and adds it to the clause r = `fly(X) :- bird(X)` because it has the best information gain among {`bird,penguin,cat`}. Then, the training set gets updated to $E^+ = \{tweety, et\}$, $E^- = \{polly\}$ in line 15-16. The negative example $polly$ is still implied by the generated clause and so is a false negative classification. The default learning of learn_rule function is finished because the best information gain of candidate literal is zero. Therefore, the FOLD-R++ function is called recursively with swapped positive and negative examples, $E^+ = \{polly\}$, $E^- = \{tweety, et\}$, to learn exceptions. In this case, an abnormal predicate {`ab0(X) :- penguin(X)`} is generated and returned as the only exception to the previous learned clause as r = `fly(X) :- bird(X), ab0(X)`. The abnormal rule {`ab0(X) :-`

[2]The FOLD-R++ toolset is available on `https://github.com/hwd404/FOLD-R-PP`.

Algorithm 1 FOLD-R++ Algorithm

Input: $target, B, E^+, E^-, ratio$ ▷ $ratio$ is the exception ratio
Output: $R = \{r_1, ..., r_n\}$ ▷ R is rule set

function FOLD-R++(E^+, E^-, L_{used})
  $R \leftarrow \emptyset$
  while $|E^+| > 0$ **do**
    $r \leftarrow$ LEARN_RULE(E^+,E^-,L_{used})
    $E^+ \leftarrow E^+ \setminus covers(r, E^+, true)$
    $R \leftarrow R \cup \{r\}$
  end while
  return R
end function
function LEARN_RULE(E^+, E^-, L_{used})
  $L \leftarrow \emptyset$
  while $true$ **do**
    $l \leftarrow$ FIND_BEST_LITERAL(E^+,E^-,L_{used})
    $L \leftarrow L \cup \{l\}$
    $r \leftarrow set_default(r, L)$
    $E^+ \leftarrow covers(r, E^+, true)$
    $E^- \leftarrow E^- \setminus covers(r, E^-, false)$
    if l is invalid or $|E^-| \leq |E^+| * ratio$ **then**
      if l is invalid **then**
        $L \leftarrow L \setminus \{l\}$
        $r \leftarrow set_default(r, L)$
      else
        $flag \leftarrow true$
      end if
      break
    end if
  end while
  if $flag$ **then**
    $AB \leftarrow$ FOLD-R++(E^-,E^+,$L_{used} + L$)
    $r \leftarrow set_exception(r, AB)$
  end if
  return r
end function

`penguin(X)`} is added to the final rule set producing the program below:

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

We next give more details of the FOLD-R++ algorithm.

3.1 Literal Selection

The literal selection process for Shakerin's FOLD-R algorithm can be summarized in Algorithm 2.

The FOLD-R algoirthm (Shakerin 2020; Shakerin, Salazar, and Gupta 2017) selects the best literal based on the weighted information gain for learning defaults, similar to the original FOLD algorithm described in (Shakerin, Salazar, and Gupta 2017). For numeric features, the FOLD-R algorithm would enumerate all the possible splits. Then, it classifies the data and compute information gain for literals for each split. The literal with the best information gain would be selected as result. In contrast, FOLD-R++ algorithm uses a new, more efficient method employing *prefix sums* to calculate the information gain based on the classification categories. In FOLD-R++, information gain for a given literal is calculated as shown in Algorithm 3.

Algorithm 2 FOLD-R Algorithm's Specialize function

function SPECIALIZE(c, E^+, E^-)
  while $size(E^-) > 0$ **do**
    $(c_1, IG_1) \leftarrow test_categorical(c, E^+, E^-)$
    $(c_2, IG_2) \leftarrow test_numeric(c, E^+, E^-)$
    if $IG_1 = 0$ & $IG_2 = 0$ **then**
      $\hat{c} \leftarrow EXCEPTION(c, E^-, E^+)$
      if $\hat{c} = null$ **then**
        $\hat{c} \leftarrow enumerate(c, E^+)$
      end if
    else
      if $IG_1 \geq IG_2$ **then**
        $\hat{c} \leftarrow c_1$
      else
        $\hat{c} \leftarrow c_2$
      end if
    end if
    $E^- \leftarrow E^- \setminus covers(\hat{c}, E^-)$
  end while
end function

Algorithm 3 FOLD-R++ Algorithm, Information Gain function

function IG(tp, fn, tn, fp)
  if $fp + fn > tp + tn$ **then**
    return $-\infty$
  end if
  $pos, neg \leftarrow tp + fp, tn + fn$
  $tot \leftarrow pos + neg$
  $result \leftarrow (\frac{tp}{tot} log_2(\frac{tp}{pos}))_{tp>0} + (\frac{fp}{tot} log_2(\frac{fp}{pos}))_{fp>0}$
  $result \leftarrow result + (\frac{tn}{tot} log_2(\frac{tn}{neg}))_{tn>0} + (\frac{fn}{tot} log_2(\frac{fn}{neg}))_{fn>0}$
  return $result$
end function

The variables tp, fn, tn, fp in Algorithm 3 for finding the information gain represent the numbers of true positive, false positive, true negative, and false negative examples, respectively. With the function above, the new approach employs the *prefix sum technique* to speed up the calculation. Only one round of classification is needed for a single feature, even with mixed types of values. The new approach to calculate the best IG and literal is summarized in Algorithm 4.

Example 2 *Given positive and negative examples, E^+, E^-, with mixed type of values on feature i, the target is to find the literal with the best information gain on the given feature. There are 8 positive examples, their values on feature i are $[1, 2, 3, 3, 5, 6, 6, b]$. And, the values on feature i of the 5 negative examples are $[2, 4, 6, 7, a]$.*

With the given examples and specified feature, the numbers

Algorithm 4 FOLD-R++ Algorithm, Best Information Gain function

Input: E^+, E^-, i
Output: $best, l$ ▷ $best$: the best IG of feature i, l: the literal with IG $best$

function BEST_INFO_GAIN(E^+, E^-, i)
  $pos, neg \leftarrow count_classification(E^+, E^-, i)$
    ▷ pos, neg are dicts that holds the # of pos / neg examples for each value
  $xs, cs \leftarrow collect_unique_values(E^+, E^-, i)$
    ▷ xs, cs are lists that holds the unique numeric and categorical values
  $xp, xn, cp, cn \leftarrow count_total(E+, E-, i)$
    ▷ (xp, xn) are the total # of pos / neg examples with numeric value, (cp, cn) are the same for categorical values.
  $xs \leftarrow sort(xs)$
  for $j \leftarrow 1$ to $size(xs)$ **do** ▷ compute the prefix sum
    $pos[xs_i] \leftarrow pos[xs_i] + pos[xs_{i-1}]$
    $neg[xs_i] \leftarrow neg[xs_i] + neg[xs_{i-1}]$
  end for
  for $x \in xs$ **do**
    $lit_dict[literal(i, \leq, x)] \leftarrow IG(pos[x], xp - pos[x] + cp, xn - neg[x] + cn, neg[x])$
    $lit_dict[literal(i, >, x)] \leftarrow IG(xp - pos[x], pos[x] + cp, neg[x] + cn, xn - neg[x])$
  end for
  for $c \in cs$ **do**
    $lit_dict[literal(i, =, x)] \leftarrow IG(pos[c], cp - pos[c] + xp, cn - neg[c] + xn, neg[c])$
    $lit_dict[literal(i, \neq, x)] \leftarrow IG(cp - pos[c] + xp, pos[c], neg[c], cn - neg[c] + xn)$
  end for
  $best, l \leftarrow best_pair(lit_dict)$
  return $best, l$
end function

of positive examples and negative examples for each unique value are counted first, which are shown as pos, neg at right side of Table 1. Then, the prefix sum arrays are calculated for computing heuristic as pos_sum, neg_sum. Table 2 show the information gain for each literal, the $literal(i, \neq, a)$ has been selected with the highest score.

3.2 Justification

Explainability is very important for some tasks like loan approval, credit card approval, and disease diagnosis system. Answer set programming provides explicit rules for how a prediction is generated compared to black box models like those based on neural networks. To efficiently justify the prediction, the FOLD-R++ outputs answer set programs that are compatible with the s(CASP) goal-directed ASP system (Arias et al. 2018).

Example 3 *The "Titanic Survival Prediction" is a classical classification challenge which contains 891 passengers as training examples and 418 passengers as testing examples and their survival based on features such as sex, age, number of siblings/spouses, number of parents/children, etc.. FOLD-R++ generates the following program with only 12 rules!:*

	examples	**ith feature value**
	$\mathbf{E}^+$	1 2 3 3 5 6 6 b
	$\mathbf{E}^-$	2 4 6 7 a

value	1	2	3	4	5	6	7	a	b
pos	1	1	2	0	1	2	0	0	1
pos_sum	1	2	4	4	5	7	7	na	na
neg	0	1	0	1	0	1	1	1	0
neg_sum	0	1	1	2	2	3	4	na	na

Table 1: Left: Examples and values on ith feature. Right: positive/negative count and prefix sum on each value

	Info Gain			
value	$\leq$ **value**	$>$ **value**	$=$ **value**	$\neq$ **value**
1	$-\infty$	-0.664	na	na
2	$-\infty$	-0.666	na	na
3	-0.619	$-\infty$	na	na
4	-0.661	$-\infty$	na	na
5	-0.642	$-\infty$	na	na
6	-0.616	$-\infty$	na	na
7	-0.661	$-\infty$	na	na
a	na	na	$-\infty$	**-0.588**
b	na	na	$-\infty$	-0.627

Table 2: The info gain on ith feature with given examples

```
(1) status(X,0):- sex(X,'male'), not ab1(X), not ab3(X),
    not ab5(X).
(2) status(X,0):- class(X,'3'), not sex(X,'male'),
    fare(X,N4), N4>23.25, not ab6(X), not ab7(X).
(3) status(X,0):- class(X,'3'), not sex(X,'male'),
    age(X,N1),N1>16.0, number_of_siblings_spouses(X,N2),
    N2=<2.0, fare(X,N4), N4>12.475, N4=<18.0,
    number_of_parents_children(X,N3), N3=<1.0,
    not ab8(X), not ab9(X).
(4) ab1(X):- number_of_siblings_spouses(X,N2), N2>2.0,
    fare(X,N4), N4>26.25,  age(X,N1), N1=<3.0, N1>2.0.
(5) ab2(X):- age(X,N1), N1>42.0, N1=<45.0,
    number_of_siblings_spouses(X,N2), N2=<0.0,
    number_of_parents_children(X,N3), N3=<0.0,
    embarked(X,'s').
(6) ab3(X):- class(X,'1'), age(X,N1), N1=<52.0,
    fare(X,N4), N4>25.587, N4=<26.55, not ab2(X).
(7) ab4(X):- number_of_parents_children(X,N3), N3=<0.0,
    age(X,N1), N1=<11.0.
(8) ab5(X):- number_of_siblings_spouses(X,N2), N2=<2.0,
    age(X,N1), N1=<12.0, not ab4(X).
(9) ab6(X):- number_of_parents_children(X,N3), N3=<0.0.
(10) ab7(X):- fare(X,N4), N4>31.275, N4=<31.387.
(11) ab8(X):- fare(X,N4), N4>15.5, N4=<17.4, age(X,N1),
     N1=<24.0.
(12) ab9(X):- age(X,N1), N1>32.0, N1=<36.0.
```

Note that `status(X,0)` means that person whose id is `X` perished, while `status(X,1)` means that person with id `X` survived. Note that we don't have any rules generated for `status(X,1)`, so we could add a rule: `status(X,1)`

`:- not status(X,0).` The above program achieves 0.94 accuracy, 0.97 precision, 0.93 recall, and 0.95 F_1 score, which is quite remarkable. Given a new data sample, the predicted answer for this data sample using the above answer set program can be efficiently produced by the s(CASP) system. The s(CASP) system can also produce a justification (a proof tree) for this prediction. Since s(CASP) is query driven, an example query such as `?- status(926, S)` which checks if passenger with id 926 perished or survived, will succeed if status of passenger 926 is indeed predicted as perished (`S` is set to 0) by the model represented by the answer set program above. The s(CASP) system can provide a proof for each query. The English description for predicates is also needed to output the proof tree in human readable format. The meaning of predicates in English is given via the `#pred` declaration, as shown below via examples:

```
#pred age(X,Y) :: 'person @(X) is of age @(Y)'.
#pred number_of_sibling_spouses(X,Y) ::
    'person @(X) had @(Y) siblings or spouses'.
#pred ab9(X) :: 'abnormal case 9 holds for @(X)'.
```

The s(CASP) system can even generate this proof in a human understandable form (Arias et al. 2020). For example, here is the justification tree generated for the passenger with id 926:

```
 ?- status(926,X).
 % QUERY:I would like to know if
    'status' holds (for 926, and X).

 ANSWER: 1 (in 4.825 ms)
 JUSTIFICATION_TREE:
 person 926 perished,
    because person 926 is male,
    and there is no evidence that 'ab1' holds (for 926),
        because there is no evidence that
        person 926 paid Var1 not equal 57.75
            for the ticket,
        and person 926 paid 57.75 for the ticket,
        and there is no evidence that
        'number_of_siblings_spouses' holds
            (for 926, and Var8).
    there is no evidence that
    abnormal case 3 holds for 926,
        because there is no evidence that
        'class' holds (for 926, and 1).
    there is no evidence that
    abnormal case 5 holds for 926,
        because there is no evidence that
        person 926 is of age Var2 not equal 30,
        and person 926 is of age 30.
The global constraints hold.
```

With the justification tree, the reason for the prediction can be easily understood by human beings. The generated ASP rule-set can also be understood by a human. In fact, s(CASP) can print the ASP rules in English, given the description of predicates in English via the `#pred` declaration explained above. If there is any unreasonable logic generated in the rule set, it can also be modified directly by the human without retraining. Thus, any bias in the data that is captured in the generated ASP rules can be corrected by the human user, and the updated ASP rule-set used for making new predictions. An example translation for two of the rules (Rules (1) and (12)) above is shown below:

```
(1)  person X perished, if
        person X is male and
        there is no evidence that 'ab1' holds (for X) and
        there is no evidence that abnormal case 3 holds
        for X and
        there is no evidence that abnormal case 5 holds
        for X.

(12) abnormal case 9 holds for X, if
        person X is of age Y and
        Y is greater than 32.0 and
        person X is of age Y and
        Y is less or equal 36.0.
```

Note that if a data sample is not predicted to hold, because the corresponding query fails on s(CASP), then a justification can be generated by asking the negation of the query. The s(CASP) system supports constructive negation, and thus negated queries can be executed in s(CASP) and their justification/proof generated just as easily as the positive queries.

4 Experiments and Performance Evaluation

In this section, we present our experiments on UCI standard benchmarks (Lichman 2013). The XGBoost Classifier is popular classification model and used as a baseline in our experiment. We used simple settings for XGBoost classifier without limiting its performance. However, XGBoost cannot deal with mixed type (numerical and categorical) of examples directly. One-hot encoding has been used for data preparation. We use precision, recall, accuracy, F_1 score, and execution time to compare the results.

FOLD-R++ does not requires any encoding before training. The original FOLD-R system used the JPL library with Java implementation. We implemented FOLD-R++ only with Python. To make inferences using the generated rules, we developed a simple ASP interpreter for our application that is part of the FOLD-R++ system. Note that the generated programs are stratified and predicates contain only variables and constants, so implementing an interpreter for such a restricted class in Python is relatively easy. However, for obtaining the justification/proof tree, or for translating the ASP rules into equivalent English text, one must use the s(CASP) system.

We also compare the FOLD-R++ algorithm with the RIPPER algorithm (Cohen 1995). RIPPER generates formulas in conjunctive normal form as an explanation of the model. Table 4 shows the comparison for two datasets from the UCI repository (Adult and Credit Card). FOLD-R++ outperforms RIPPER on all categories except precision. Most significantly, FOLD-R++ generates much smaller number of rules. Computation time for FOLD-R++ is also a lot less.

As discussed earlier, the time complexity for computing information gain on a feature is significantly reduced in FOLD-R++ due to the use of prefix-sum. Therefore, we obtain a rather large improvements in efficiency. For the credit

		XGBoost					FOLD-R++				
DataSet	Shape	Prec.	Rec.	Acc.	F1	Time (ms)	Prec.	Rec.	Acc.	F1	Time (ms)
acute	(120, 7)	**1**	1	**1**	**1**	35	0.99	1	0.99	0.99	**2.5**
autism	(704, 18)	**0.97**	**0.98**	**0.98**	0.97	76	0.95	0.96	0.97	0.97	**47**
breast-w	(699, 10)	0.95	0.97	0.96	0.96	78	**0.96**	0.97	0.96	**0.97**	**28**
cars	(1728, 7)	**1**	1	**1**	**1**	77	0.98	1	0.97	0.98	**48**
credit-a	(690, 16)	**0.85**	0.83	**0.83**	0.83	368	0.84	**0.92**	0.79	**0.84**	**100**
ecoli	(336, 9)	0.76	0.76	0.62	0.68	165	**0.96**	**0.95**	**0.94**	**0.95**	**28**
heart	(270, 14)	**0.80**	**0.81**	0.83	0.81	112	0.79	0.79	0.83	0.81	**44**
ionosphere	(351, 35)	0.88	0.86	**0.96**	0.90	1,126	**0.92**	**0.93**	0.94	**0.93**	**392**
kidney	(400, 25)	0.98	0.98	0.98	0.98	126	**0.99**	**1**	0.98	**0.99**	**27**
kr vs. kp	(3196, 37)	0.99	0.99	0.99	0.99	**210**	0.99	0.99	0.99	0.99	361
mushroom	(8124, 23)	1	1	1	1	**378**	1	1	1	1	476
sonar	(208, 61)	0.53	0.54	**0.84**	0.65	1,178	**0.78**	**0.81**	0.75	**0.78**	**419**
voting	(435, 17)	0.95	0.94	**0.95**	0.94	49	0.95	0.94	0.94	0.94	**16**
adult	(32561, 15)	**0.86**	**0.88**	0.94	**0.91**	274,655	0.84	0.86	**0.95**	0.90	**10,069**
credit card	(30000, 24)	-	-	-	-	-	**0.82**	**0.83**	**0.96**	**0.89**	**21,349**

Table 3: Evaluation of FOLD-R++ on UCI Datasets

Data	Adult		Credit card	
Shape	(32561 , 15)		(30000 , 24)	
Algo	RIPPER	FOLDR++	RIPPER	FOLDR++
Acc.	0.70	**0.84**	0.77	**0.82**
Prec	**0.96**	0.86	**0.87**	0.83
Rec	0.63	**0.95**	0.83	**0.96**
F1	0.76	**0.90**	0.85	**0.89**
# Rules	46.9	**16.7**	38.4	**19.1**
Time	59.5s	**10.1s**	47.4s	**27.9s**

Table 4: Comparison with RIPPER Algorithm

dataset, a dataset with only 690 instances, the new FOLD-R++ algorithm is hundreds times faster than the original FOLD-R. All the learning experiments have been conducted on a desktop with Intel i5-10400 CPU @ 2.9GHz and 32 GB ram. To measure performance metrics, we conducted 10-fold cross-validation on each dataset and the average of accuracy, precision, recall, F_1 score and execution time have been presented. Table 3 reports the performance metrics and execution time on each dataset compared with the baseline model. The best performer is highlighted with boldface font.

The XGBoost Classifier employs decision tree ensemble method for classification task and provides quite decent performance. FOLD-R++ almost always spends less time to finish learning compared to XGBoost classifier, especially for the large dataset Adult income census. For most of the datasets, FOLD-R++ can achieve equivalent scores. FOLD-R++ achieves much higher scores on e-coli and sonar datasets. For the credit card dataset, the baseline XGBoost model failed training due to 32 GB memory limitation, but FOLD-R++ still finished training quite efficiently.

5 Related Work

ALEPH (Srinivasan 2001) is one of the most popular ILP system, which induces theories by using bottom-up generalization search. However, it cannot deal with numeric features and its specialization step is manual, there is no automation option. Takemura and Inoue's method (Takemura and Inoue 2021) relies on tree-ensembles to generate explainable rule sets with pattern mining techniques. Its performance depends on the tree-ensemble model. Additionally, it may not be scalable due to its computational time complexity that is exponential in the number of valid rules.

A survey of ILP can be found in (Muggleton et al. 2012). Rule extraction from statistical Machine Learning models has been a long-standing goal of the community. The rule extraction algorithms from machine learning models are classified into two categories: 1) Pedagogical (i.e., learning symbolic rules from black-box classifiers without opening them) 2) Decompositional (i.e., to open the classifier and look into the internals). TREPAN (Craven and Shavlik 1995) is a successful pedagogical algorithm that learns decision trees from neural networks. SVM+Prototypes (Núñez, Angulo, and Català 2002) is a decompositional rule extraction algorithm that makes use of KMeans clustering to extract rules from SVM classifiers by focusing on support vectors. Another rule extraction technique that is gaining attention recently is "RuleFit" (Friedman, Popescu, and others 2008). RuleFit learns a set of weighted rules from ensemble of shallow decision trees combined with original features. In ILP community also, researchers have tried to combine statistical methods with ILP techniques. Support Vector ILP (Muggleton et al. 2005) uses ILP hypotheses as kernel in dual form of the SVM algorithm. kFOIL (Landwehr et al. 2006) learns an incremental kernel for SVM algorithm using a FOIL style specialization. nFOIL (Landwehr, Kersting, and Raedt 2005) integrates the Naive-Bayes algorithm with FOIL. The advantage of our research over all of the above mentioned research work is that we generate answer set programs containing negation-as-failure that correspond closely to the human thought process. Thus, the descriptions are more concise. Second it is scalable thanks to the greedy

nature of our clause search.

6 Conclusions and Future Work

In this paper we presented an efficient and highly scalable algorithm, FOLD-R++, to induce default theories represented as an answer set program. The resulting answer set program has good performance wrt prediction and justification for the predicted classification. In this new approach, unlike other methods, the encoding for data is not needed anymore and no information from training data is discarded. Compared with the popular classification system XGBoost, our new approach has similar performance in terms of accuracy, precision, recall, and F1-score, but better training efficiency. In addition, the FOLD-R++ algorithm produces an explainable model. Predictions made by this model can be computed efficiently and their justification automatically produced using the s(CASP) system.

Acknowledgement

Authors gratefully acknowledge support from NSF grants IIS 1718945, IIS 1910131, IIP 1916206, and from Amazon Corp, Atos Corp and US DoD. Views expressed are authors' own and not of the funding agencies. We are grateful to Joaquin Arias and the s(CASP) team for their work on providing facilities for generating the justification tree and English encoding of rules in s(CASP).

References

Arias, J.; Carro, M.; Salazar, E.; Marple, K.; and Gupta, G. 2018. Constraint answer set programming without grounding. *Theory and Practice of Logic Programming* 18(3-4):337–354.

Arias, J.; Carro, M.; Chen, Z.; and Gupta, G. 2020. Justifications for goal-directed constraint answer set programming. In *Proceedings 36th International Conference on Logic Programming (Technical Communications)*, volume 325 of *EPTCS*, 59–72.

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge, New York, Melbourne: Cambridge University Press.

Chen, T., and Guestrin, C. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD*, KDD '16, 785–794.

Cohen, W. W. 1995. Fast effective rule induction. In Prieditis, A., and Russell, S. J., eds., *Machine Learning, Proceedings of the Twelfth International Conference on Machine Learning*, 115–123. Morgan Kaufmann.

Craven, M. W., and Shavlik, J. W. 1995. Extracting tree-structured representations of trained networks. In *Proceedings of the 8th International Conference on Neural Information Processing Systems*, NIPS'95, 24–30. Cambridge, MA, USA: MIT Press.

Friedman, J. H.; Popescu, B. E.; et al. 2008. Predictive learning via rule ensembles. *The Annals of Applied Statistics* 2(3):916–954.

Gelfond, M., and Kahl, Y. 2014. *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press.

Gunning, D. 2015. Explainable artificial intelligence (xai).

Landwehr, N.; Passerini, A.; Raedt, L. D.; and Frasconi, P. 2006. kFOIL: Learning simple relational kernels. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, MA, USA*, 389–394.

Landwehr, N.; Kersting, K.; and Raedt, L. D. 2005. nFOIL: Integrating naïve bayes and FOIL. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, 795–800.

Lichman, M. 2013. UCI, Machine Learning Repository, `http://archive.ics.uci.edu/ml`.

Muggleton, S.; Lodhi, H.; Amini, A.; and Sternberg, M. J. E. 2005. Support vector inductive logic programming. In Hoffmann, A.; Motoda, H.; and Scheffer, T., eds., *Discovery Science*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Muggleton, S.; Raedt, L.; Poole, D.; Bratko, I.; Flach, P.; Inoue, K.; and Srinivasan, A. 2012. Ilp turns 20. *Mach. Learn.* 86(1):3–23.

Muggleton, S. 1991. Inductive logic programming. *New Gen. Comput.* 8(4).

Núñez, H.; Angulo, C.; and Català, A. 2002. Rule extraction from support vector machines. In *In Proceedings of European Symposium on Artificial Neural Networks*, 107–112.

Plotkin, G. D. 1971. A further note on inductive generalization, in machine intelligence, volume 6, pages 101-124.

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

Sakama, C. 2005. Induction from answer sets in nonmonotonic logic programs. *ACM Trans. Comput. Log.* 6(2):203–231.

Shakerin, F.; Salazar, E.; and Gupta, G. 2017. A new algorithm to automate inductive learning of default theories. *TPLP* 17(5-6):1010–1026.

Shakerin, F. 2020. *Logic Programming-based Approaches in Explainable AI and Natural Language Processing*. Ph.D. Dissertation. Department of Computer Science, The University of Texas at Dallas.

Srinivasan, A. 2001. *The Aleph Manual,* `https://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html`.

Takemura, A., and Inoue, K. 2021. Generating explainable rule sets from tree-ensemble learning methods by answer set programming. *Electronic Proceedings in Theoretical Computer Science* 345:127–140.

Wikipedia contributors. 2021. Prefix sum Wikipedia, the free encyclopedia. Online; accessed 5 October, 2021.

Zeng, Q.; Patel, J. M.; and Page, D. 2014. Quickfoil:

Scalable inductive logic programming. *Proc. VLDB Endow.* 8(3):197–208.