TPLP **22** (5): 658–677, 2022.
doi:10.1017/S1471068422000205 First published online 1 July 2022

FOLD-RM: A Scalable, Efficient, and Explainable Inductive Learning Algorithm for Multi-Category Classification of Mixed Data

HUADUO WANG, FARHAD SHAKERIN and GOPAL GUPTA
The University of Texas at Dallas, Richardson, USA
(*e-mails:* `huaduo.wang@utdallas.edu`, `farhad.shakerin@utdallas.edu`, `gupta@utdallas.edu`)

submitted 15 May 2022; accepted 08 June 2022

Abstract

FOLD-RM is an automated inductive learning algorithm for learning default rules for mixed (numerical and categorical) data. It generates an (explainable) answer set programming (ASP) rule set for *multi-category classification* tasks while maintaining efficiency and scalability. The FOLD-RM algorithm is competitive in performance with the widely used, state-of-the-art algorithms such as XGBoost and multi-layer perceptrons, however, unlike these algorithms, the FOLD-RM algorithm produces an explainable model. FOLD-RM outperforms XGBoost on some datasets, particularly large ones. FOLD-RM also provides human-friendly explanations for predictions.

KEYWORDS: explainable AI, data mining, inductive logic programming, machine learning

1 Introduction

Dramatic success of machine learning has led to an avalanche of applications of artificial intelligence (AI). However, the effectiveness of these systems is limited by the machines' current inability to explain their decisions to human users. That is mainly because 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 problem of explaining (or justifying) a model's decision to its human user is referred to as the model interpretability problem. The subfield is referred to as explainable AI (XAI).

The inductive logic programming (ILP) learning problem is the problem of searching for a set of logic programming clauses from which the training examples can be deduced. ILP provides an excellent solution for XAI. ILP is a thriving field and a large number of such clause search algorithms have been devised as described by Muggleton *et al.* (2012) and Cropper and Dumancic (2020). The search in these ILP algorithms 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* (NAF) into the hypothesis, as explained in the book by Baral (2003). A survey of bottom-up ILP systems and their shortcomings has been compiled by Sakama (2005). In contrast, top-down approach starts with the most general clauses and then specializes them. A top-down algorithm guided by heuristics is better suited for large-scale and/or noisy datasets, as explained by Zeng *et al.* (2014).

The FOIL algorithm by Quinlan is a popular top-down ILP algorithm that learns a logic program. The FOLD algorithm by Shakerin *et al.* (2017) is a novel top-down algorithm that learns default rules along with exception(s) that closely model human thinking. It first learns default predicates that cover positive examples while avoiding covering negative examples. Then it swaps the covered positive examples and negative examples and calls itself recursively to learn the exception to the default. It repeats this process to learn exceptions to exceptions, exceptions to exceptions to exceptions, and so on. The FOLD-R++ algorithm by Wang and Gupta (2022) is a new scalable ILP algorithm that builds upon the FOLD algorithm to deal with the efficiency and scalability issues of the FOLD and FOIL algorithms. It introduces the prefix sum computation and other optimizations to speed up the learning process while providing human-friendly explanation for its prediction using the s(CASP) answer set programming system (ASP) of Arias *et al.* (2018). However, all these algorithms focus on binary classification tasks, and cannot deal with multi-category classification tasks. Note that a binary classification task checks whether a data record is a member of a given class or not, for example, does a given creature fly or not fly? In multi-category classification, there can be multiple membership classes, for example, a given creature's habitat can be predicted to be one of desert, mountain, plain, salt water, or fresh water (see the textbook by Bishop (2006)).

In this paper we propose a new ILP learning algorithm called FOLD-RM for multi-category classification that builds upon the FOLD-R++ algorithm. FOLD-RM also provides native explanations for prediction without external libraries or tools. Our experimental results indicates that the FOLD-RM algorithm is comparable in performance to traditional, popular machine learning algorithms such as XGBoost by Chen and Guestrin (2016) and multi-layer perceptrons (MLP) described in the book by Aggarwal (2018). In most cases, FOLD-RM outperforms them in execution efficiency. Of course, neither XGBoost nor MLP are interpretable.

Note that the term model in the field of machine learning and logic programming have different meanings. We use the term model in this paper in machine learning sense. Thus, the answer set program generated by our FOLD-RM algorithm *is the model that we learn* in the sense of machine learning. We use the term *answer set* in this paper to refer to stable models of answer set programs, where a model means assignment of truth values to program predicates that make the program true. Note also that we use the terms clause and rule interchangeably in this paper.

2 Background

2.1 Inductive logic programming

ILP as described in Muggleton (1991) is a subfield of machine learning that learns models in the form of logic programming clauses comprehensible to humans. This problem is formally defined as:

Given

1. A background theory B, in the form of an extended logic program, that is, clauses of the form $h \leftarrow l_1, ..., l_m,\ not\ l_{m+1}, ...,\ not\ l_n$, where $h, l_1, ..., l_n$ are positive literals and *not* denotes NAF as described in Baral (2003). For reasons of efficiency, we restrict B to be stratified (stratified logic programs are explained in the book by Gelfond and Kahl (2014)).
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 θ-subsumption described in Plotkin (1971) (for more details, see the paper by Cropper and Dumancic (2020)). The hypothesis language L is also assumed to be stratified.

Find a set of clauses H such that:

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

The target predicate is the predicate whose definition we want to learn as a stratified normal logic program. The positive and negative examples are grounded target predicates, that is, suppose we want to learn the concept of which creatures can *fly*, then we will give positive examples $E^+ = \{$`fly(tweety), fly(sam), ...`$\}$ and negative examples $E^- = \{$`fly(kitty), fly(polly), ...`$\}$, where `tweety`, `sam`, ..., are names of creatures that can fly, and `kitty`, `polly`, ..., are names of creatures that cannot fly.

Note that the reason for restricting to stratified normal logic programs is that we can realize a simple and efficient ASP interpreter in the FOLD-RM system code for the training process. If we allowed for non-stratified programs, the training process will have to invoke a full-fledged ASP interpreter during the training and testing process, resulting in significant inefficiency. Considering non-stratified programs is part of our future research plan. We restrict ourselves to function-free predicates, that is, we allow only datalog rules, again, for reasons of efficiency.

2.2 Default rules

Default logic proposed by Reiter (1980) is a non-monotonic logic to formalize commonsense reasoning. A default D is an expression of the form

$$\frac{A : \mathbf{M}B}{\Gamma},$$

which states that the conclusion Γ can be inferred if pre-requisite A holds and B is justified. $\mathbf{M}B$ stands for "it is consistent to believe B" as explained in the book by Gelfond and Kahl (2014). Normal logic programs can encode a default quite elegantly. A default of the form:

$$\frac{\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n : \mathbf{M}\neg\beta_1, \mathbf{M}\neg\beta_2 \ldots \mathbf{M}\neg\beta_m}{\gamma},$$

can be formalized as the following normal logic program rule:

$$\gamma \texttt{ :- } \alpha_1, \alpha_2, \ldots, \alpha_n, \texttt{not } \beta_1, \texttt{not } \beta_2, \ldots, \texttt{not } \beta_m.$$

where α's and β's are positive predicates and `not` represents NAF (under the stable model semantics as described in Baral (2003)). We call such rules default rules. Thus, the default $\frac{bird(X):M\neg penguin(X)}{fly(X)}$ will be represented as the following ASP-coded default rule:

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

We call `bird(X)`, the condition that allows us to jump to the default conclusion that `X` can fly, as the *default part* of the rule, and `not penguin(X)` as the *exception part* of the rule.

Default rules closely represent the human thought process (i.e. frequently used in commonsense reasoning). FOLD-R and FOLD-R++ learn default rules represented as answer set programs. Note that the programs currently generated are stratified normal logic programs, however, we eventually hope to learn non-stratified answer set programs too as in the work of Shakerin and Gupta (2018) and Shakerin (2020). Hence, we continue to use the term answer set program for a normal logic program in this paper. An advantage of learning default rules is that we can distinguish between exceptions and noise as explained by Shakerin *et al.* (2017) and Shakerin (2020). The introduction of (nested) exceptions, or abnormal predicates, in a default rule increases coverage of the data by that default rule. A single rule can now cover more examples which results in reduced number of generated rules. The equivalent program without the abnormal predicates will have many more rules if the abnormal predicates calls are fully expanded.

2.3 Classification problems

Classification problems are either binary or multi-category.

1. Binary classification is the task of classifying the elements of a set into two groups on the basis of a classification rule. For example, a specific patient (given a set of patients) has a particular disease or not, or a particular manufactured article (in a set of manufactured articles) will pass quality control or not. Details can be found in the book by Bishop (2006).
2. Multi-category or multinomial classification is the problem of classifying instances into one of three or more classes. For example, an animal can be predicted to have one of the following habitats: sea water, fresh water, desert, mountain, or plains. Again, details can be found in the book by Bishop (2006).

3 The FOLD-R++ algorithm

The FOLD-R++ algorithm by Wang and Gupta (2022) is a new ILP algorithm for binary classification that is built upon the FOLD algorithm of Shakerin *et al.* (2017). Our FOLD-RM algorithm builds upon the FOLD-R++ algorithm. FOLD-R++ increases the efficiency and scalability of the FOLD algorithm. The FOLD-R++ algorithms divides features into two categories: categorical features and numerical features. For a categorical

Algorithm 1 FOLD-R++ Algorithm: Information Gain function

Input: tp, fn, tn, fp: the number of E_{tp}, E_{fn}, E_{tn}, E_{fp} implied by literal
Output: information gain

```
 1: function F($a, b$)
 2:     if $a = 0$ then
 3:         return 0
 4:     end if
 5:     return $a \cdot \log_2(\frac{a}{a+b})$
 6: end function
 7: function IG($tp, fn, tn, fp$)
 8:     if $fp + fn > tp + tn$ then
 9:         return $-\infty$
10:     end if
11:     return $\frac{1}{tp+fp+tn+fn} \cdot (\mathrm{F}(tp, fp) + \mathrm{F}(fp, tp) + \mathrm{F}(tn, fn) + \mathrm{F}(fn, tn))$
12: end function
```

feature, all the values in the feature would be considered as categorical values even though some of them are numbers. For categorical features, the FOLD-R++ algorithm only generates equality or inequality literals. For numerical features, the FOLD-R++ algorithm would try to read all the values as numbers, converting them to categorical values if conversion to numbers fails. FOLD-R++ additionally generates numerical comparison ($\leq$ and $>$) literals for numerical values. For a mixed type feature that contains both categorical values and numerical values, the FOLD-R++ algorithm treats them as numerical features.

The FOLD-R++ algorithm employs information gain (IG) heuristic to guide literal selection during the learning process. It uses a simplified calculation process for IG by using the number of true positive, false positive, true negative, and false negative examples that a literal can imply. The IG for a given literal is calculated as shown in Algorithm 1.

The goal of the ILP algorithm is to find an answer set program whose answer set has all the positive examples and none of the negative examples. Our algorithm incrementally learns this program using the IG heuristic. The IG heuristic allows us to refine our program incrementally, that is, the answer set of the program after each refinement step has more and more positive examples included and fewer and fewer of the negative ones.

The comparison between two numerical values or two categorical values in FOLD-R++ is straightforward, as commonsense would dictate, that is, two numerical (*resp.* categorical) values are equal if they are identical, else they are unequal. However, a different assumption is made to compare a numerical value and a categorical value in FOLD-R++. The equality between a numerical value and a categorical value is always false, and the inequality between a numerical value and a categorical value is always true. Additionally, numerical comparison ($\leq$ and $>$) between a numerical value and a categorical value is always false. An example is shown in Table 1 (left), while an evaluation example for a given literal, $literal(i, >, 4)$, based on the comparison assumption is shown in Table 1 (right). Given $\mathrm{E}^+ = \{1, 2, 2, 4, 5, x, x, y\}$, $\mathrm{E}^- = \{1, 3, 4, y, y, y, z\}$, and $literal(i, >, 4)$, the true positive example E_{tp}, false negative examples E_{fn}, true negative examples E_{tn}, and false positive examples E_{fp} implied by the literal are $\{5\}$, $\{1, 2, 2, 4, x, x, y\}$, $\{1, 3, 4, y, y, y, z\}$,

Table 1. *Left: Comparisons between a numerical value and a categorical value. Right: Evaluation and count for literal*$(i, >, 4)$.

comparison	**evaluation**
5 = 'k'	False
5 ≠ 'k'	True
5 ≤ 'k'	False
5 > 'k'	False

	ith feature values	**count**
$\mathbf{E^+}$	1 2 2 4 5 x x y	**8**
$\mathbf{E^-}$	1 3 4 y y y z	**7**
$\mathbf{E_{tp(i,>,4)}}$	5	**1**
$\mathbf{E_{fn(i,>,4)}}$	1 2 2 4 x x y	**7**
$\mathbf{E_{tn(i,>,4)}}$	1 3 4 y y y z	**7**
$\mathbf{E_{fp(i,>,4)}}$	∅	**0**

∅ respectively. Then, the IG of literal$(i, >, 4)$ is calculated $IG_{(i,>,4)}(1, 7, 7, 0) = -0.647$ through Algorithm 1.

The FOLD-R++ algorithm starts with the clause `p(...) :- true.`, where `p(...)` is the target predicate to learn. It specializes this clause by adding literals to its body during the inductive learning process. It selects a literal to add that maximizes IG. The literal selection process is summarized in Algorithm 2. In line 2, *pos* and *neg* are dictionaries that hold, respectively, the numbers of positive and & negative examples for each unique value. In line 3, xs and & cs are lists that hold, respectively, the unique numerical and categorical values. In line 4, xp and & xn are the total number of, respectively, positive and negative examples with numerical values; cp and cn are the same for categorical values. In line 11, the IG of literal$(i, \leq, x)$ is calculated by taking the parameters $pos[x]$ as the number of true positive examples, $xp - pos[x] + cp$ as the number of false negative examples, $xn - neg[x] + cn$ as the number of true negative examples, and $neg[x]$ as the number of false positive examples. After computing the prefix sum in line 6, $pos[x]$ holds the total number of positive examples that has a value less than or equal to x. Therefore, $xp-pos[x]$ represents the total number of positive examples that have a value greater than x. cp, the total number of positive examples that have a categorical value, is added to the number of false negative examples because of the assumption that numerical comparison between a numerical value and a categorical value is always false. The negative examples that have a value greater than x or a categorical value would be evaluated as false by literal$(i, \leq, x)$, so $xn - neg[x]$ is added as true negative parameter. And, cn, the total number of negative examples that has a categorical value, is added to true negative parameter. The expression $neg[x]$ means the number of negative examples that have the value less than or equal to x; $neg[x]$ is added as false positive parameter because the evaluations of these examples by literal$(i, \leq, x)$ are true. The IG calculation processes of other literals also follows the comparison assumption mentioned above. Finally, the `best_info_gain` function returns the best score on IG and the corresponding literal except the literals that have been used in current rule-learning process. For each feature, we compute the best literal, then the `find_best_literal` function returns the best literal among this set of best literals.

Example 1
Given positive and negative examples in Table 2, E$^+$, E$^-$, with mixed type of values on ith feature, the target is to find the literal with the best IG on the given feature. There are 8 positive examples, their values on ith feature are $[1, 2, 2, 4, 5, x, x, y]$, and the values on ith feature of the 7 negative examples are $[1, 3, 4, y, y, y, z]$.

Algorithm 2 FOLD-R++ Algorithm, Find Best Literal function

Input: E^+: positive examples, E^-: negative examples, L_{used}: used literals

Output: *best_lit*: the best literal that provides the most information

```
1:  function BEST_INFO_GAIN($E^+, E^-, i, L_{used}$)
2:      $pos, neg \leftarrow count\_classification(E^+, E^-, i)$
3:      $xs, cs \leftarrow collect\_unique\_values(E^+, E^-, i)$
4:      $xp, xn, cp, cn \leftarrow count\_total(E^+, E^-, i)$
5:      $xs \leftarrow couting\_sort(xs)$
6:      for $j \leftarrow 1$ to $size(xs)$ do    ▷ compute prefix sum for $E^+$ & $E^-$ numerical values
7:          $pos[xs_j] \leftarrow pos[xs_j] + pos[xs_{j-1}]$
8:          $neg[xs_j] \leftarrow neg[xs_j] + neg[xs_{j-1}]$
9:      end for
10:     for $x \in xs$ do
11:         $lit\_dict[literal(i, \leq, x)] \leftarrow \mathrm{IG}(pos[x], xp - pos[x] + cp, xn - neg[x] + cn, neg[x])$
12:         $lit\_dict[literal(i, >, x)] \leftarrow \mathrm{IG}(xp - pos[x], pos[x] + cp, neg[x] + cn, xn - neg[x])$
13:     end for
14:     for $c \in cs$ do
15:         $lit\_dict[literal(i, =, x)] \leftarrow \mathrm{IG}(pos[c], cp - pos[c] + xp, cn - neg[c] + xn, neg[c])$
16:         $lit\_dict[literal(i, \neq, x)] \leftarrow \mathrm{IG}(cp - pos[c] + xp, pos[c], neg[c], cn - neg[c] + xn)$
17:     end for
18:     $best, l \leftarrow best\_pair(lit\_dict, L_{used})$
19:     return $best, l$    ▷ return the best info gain and its corresponding literal
20: end function
21: function FIND_BEST_LITERAL($E^+, E^-, L_{used}$)
22:     $best\_ig, best\_lit \leftarrow -\infty, invalid$
23:     for $i \leftarrow 1$ to $N$ do    ▷ $N$ is the number of features
24:         $ig, lit \leftarrow$ BEST_INFO_GAIN($E^+$,$E^-$,i,$L_{used}$)
25:         if $best\_ig < ig$ then
26:             $best\_ig, best\_lit \leftarrow ig, lit$
27:         end if
28:     end for
29:     return $best\_lit$
30: end function
```

With the given examples and specified feature, the number of positive examples and negative examples for each unique value are counted first, which are shown as pos, neg on right side of Table 2. Then, the prefix sum arrays are calculated for computing heuristic as psum$^+$, psum$^-$. Table 3 shows the IG for each literal, the literal$(i, =, x)$ has been selected with the highest score.

4 The FOLD-RM algorithm

The FOLD-R++ algorithm performs binary classification. We generalize the FOLD-R++ algorithm to perform multi-category classification. The generalized algorithm is called FOLD-RM. The FOLD-R++ algorithm is summarized in Algorithm 3. The FOLD-R++ algorithm generates an ASP rule set, in which all the rules have the same rule head.

Table 2. *Left: Examples and values on i^{th} feature. Right: positive/negative count and prefix sum on each value*

	i^{th} feature values							
E^+	1	2	2	4	5	x	x	y
E^-	1	3	4	y	y	y	z	

value	1	2	3	4	5	x	y	z
pos	1	2	0	1	1	2	1	0
$psum^+$	1	3	3	4	5	N/A	N/A	N/A
neg	1	0	1	1	0	0	3	1
$psum^-$	1	1	2	3	3	N/A	N/A	N/A

Table 3. *The info gain on i^{th} feature with given examples*

	Info Gain							
value	1	2	3	4	5	x	y	z
$\leq$ **value**	$-\infty$	-0.655	-0.686	-0.688	-0.672	N/A	N/A	N/A
$>$ **value**	-0.667	$-\infty$	-0.682	-0.647	$-\infty$	N/A	N/A	N/A
$=$ **value**	N/A	N/A	N/A	N/A	N/A	-0.598	$-\infty$	$-\infty$
$\neq$ **value**	N/A	N/A	N/A	N/A	N/A	$-\infty$	-0.631	-0.637

An example covered by any rule in the set would imply the rule head is true. The FOLD-R++ algorithm generates a model by learning one rule at a time. Ruling out the already covered example in line 9 after learning a rule would help select better literal for remaining examples. In the rule-learning process, the best literal would be selected according to the useful information it can provide for current training examples (line 17) till the literal selection fails. If the ratio of false positive examples to true positive examples drops below the threshold *ratio* in line 22, it would next learn exceptions by swapping residual positive and negative examples and calling itself recursively (line 26). Any examples that cannot be covered by the selected literals would be ruled out in line 20, 21. The *ratio* in line 22 represents the upper bound on the number of true positive examples to the number of false positive examples implied by the default part of a rule. It helps speed up the training process and reduces the number of rules learned.

Generally, avoiding covering negative examples by adding literals to the default part of a rule will reduce the number of positive examples the rule can imply. Explicitly activating the exception learning procedure (line 26 in Algorithm 3) could increase the number of positive example a rule can cover while reducing the total number of rules generated. As a result, the interpretability is increased due to fewer rules being generated.

The FOLD-RM algorithm performs multi-category classification. It generates rules that it can learn for each category. If an example cannot be implied by any rule in the learned rule set, it means the model fails to classify this example. The FOLD-RM algorithm, summarized in Algorithm 4, first finds a target literal that represents the category with most examples among the current training set (line 4). It next splits the training set into positive and negative examples based on the target literal (line 5). Then, it learns a rule to cover the target category (line 6) by calling the `learn_rule` function of the FOLD-R++ algorithm. The already covered examples would be ruled out from the training set in line 11, and the rule head would be changed to the target literal in line 12. However, there is a difference between the outputs of FOLD-RM and FOLD-R++.

Algorithm 3 FOLD-R++ Algorithm

```
Input: $E^+$: positive examples, $E^-$: negative examples
    ▷ Global Parameters: $target$, $B$: background knowledge, $ratio$: exception ratio
Output: $R = \{r_1, ..., r_n\}$: a set of defaults rules with exceptions
 1: function FOLD_RPP($E^+, E^-, L_{used}$)            ▷ $L_{used}$: used literals, initially empty
 2:     $R \leftarrow \emptyset$
 3:     while $|E^+| > 0$ do
 4:         $r \leftarrow$ LEARN_RULE($E^+$, $E^-$, $L_{used}$)
 5:         $E_{FN} \leftarrow covers(r, E^+, false)$ ▷ $E_{FN}$: false negative examples implied by rule $r$
 6:         if $|E_{FN}| = |E^+|$ then
 7:             break
 8:         end if
 9:         $E^+ \leftarrow E_{FN}$                     ▷ rule out the already covered examples
10:         $R \leftarrow R \cup \{r\}$
11:     end while
12:     return $R$
13: end function
14: function LEARN_RULE($E^+, E^-, L_{used}$)
15:     $L \leftarrow \emptyset$
16:     while $true$ do
17:         $l \leftarrow$ FIND_BEST_LITERAL($E^+$, $E^-$, $L_{used}$)
18:         $L \leftarrow L \cup \{l\}$
19:         $r \leftarrow set\_default(r,\ L)$                ▷ set default part of rule $r$ as $L$
20:         $E^+ \leftarrow covers(r,\ E^+,\ true)$
21:         $E^- \leftarrow covers(r,\ E^-,\ true)$
22:         if $l$ is invalid or $|E^-| \leq |E^+| * ratio$ then
23:             if $l$ is invalid then
24:                 $r \leftarrow set\_default(r,\ L \setminus\ \{l\})$   ▷ remove the invalid literal $l$ from rule $r$
25:             else
26:                 $AB \leftarrow$ FOLD_RPP($E^-$, $E^+$, $L_{used} + L$)   ▷ learn exception rules for $r$
27:                 $r \leftarrow set\_exception(r,\ AB)$        ▷ set exception part of rule $r$ as $AB$
28:             end if
29:             break
30:         end if
31:     end while
32:     return $r$                                  ▷ the head of rule $r$ is $target$
33: end function
```

Unlike FOLD-R++, the output of FOLD-RM is a textually ordered answer set program, which means a rule is checked only if all the rules before it did not apply. The FOLD-RM system is publicly available at https://github.com/hwd404/FOLD-RM.

Note that for learning each rule, FOLD-RM (Algorithm 4) chooses the target predicate by finding the label value with the most examples in the remaining training examples and sets it as the target predicate for this rule. In other words, the target predicate is the "most popular" label value. The names of the predicates are the names of features in the data. The head predicate and predicates in rule body each have exactly two arguments.

The first argument is a reference to the data record itself. For the target predicate, the second argument is the predicted label for that record, while for predicates in the body, the second argument is used to extract the appropriate feature value for that record. The abnormal predicates only take one argument, namely, the data record itself. For example, consider:

```
class(X,2) :- condition(X, s), not ab5(X).
ab5(X) :- not steel(X,r), not enamelability(X,2).
```

The first rule says that the predicted class of data record X is '2 if the condition feature of X has value 's, and abnormal case ab5 does not apply. ab5(X) is an abnormal case predicate and has only one argument. It says that the record X should not be predicted to have class value '2, if the value of steel feature is not 'r, and the value of enamelability feature is not '2.

4.1 Algorithmic complexity

Next, we analyze the complexity of the FOLD-RM algorithm. If M is the number of examples and N is the number of features, it is easy to see that the time complexity of finding the best literal (Algorithm 2) is $O(NM)$. We assume that counting sort (complexity $O(M)$) with a pre-sorted list is used at line 5 in Algorithm 2. The worst case in the FOLD-RM algorithm arises when each generated rule only covers one example and each literal only excludes one non-target example. Therefore, in the worst case there will be $O(M^2)$ literals chosen in total. The worst case time complexity of the FOLD-RM algorithm (Algorithm 4) can be calculated to be $O(NM^3)$. However, this is a theoretical upper bound. The actual learning process is really efficient because the heuristics we employ helps select very effective literals, reducing the number of iterations in the algorithm.

One can also prove that the FOLD-RM algorithm always terminates. The `fold_rm` function calls the `learn_rule` function to induce a rule that can cover at least one 'most popular' remaining example till all the examples have been covered or the learned rule fails to cover any 'most popular' example. The loop in the `fold_rm` function iterates at most $|E|$ times while excluding the already covered examples. The `learn_rule` function refines the rule with a given target by adding the best literal to the rule body. By adding literals to the rules, the numbers of true positive and false positive examples the rule implies can only monotonically decrease. The learned valid literal excludes at least one false positive example that the rule implies. So, the loop in the `learn_rule` function iterates at most $|E^-|$ times. When the $|E^-| < |E^+| * ratio$ condition is met, the `fold_rpp` function is called to learn exception rules for the current default rule. Similar to the `fold_rm` function, the `fold_rpp` function iterates at most $|E^+|$ times. Also, there are only finite for-loops inside the `find_best_literal` function. Therefore, we can conclude that the FOLD-RM algorithm will always terminate.

4.2 An illustrative example

We illustrate FOLD-RM, next, with a simple example.

Algorithm 4 FOLD-RM Algorithm

```
Input: $E$: examples, $B$: background knowledge, $ratio$: exception ratio
Output: $R = \{r_1, ..., r_n\}$: a set of defaults rules with exceptions
function FOLD_RM($E$)
    $R \leftarrow \emptyset$
    while $|E| > 0$ do
        $l \leftarrow$ MOST($E$)                    ▷ $l$: most popular target literal as the learning target
        $E^+, E^- \leftarrow$ SPLIT_BY_LITERAL($E, l$)
        $r \leftarrow$ LEARN_RULE($E^+$, $E^-$, $\emptyset$)
        $E_{FN} \leftarrow covers(r,\ E^+,\ false)$
        if $|E_{FN}| = |E^+|$ then
            break
        end if
        $E \leftarrow E^+ \cup E_{FN}$                    ▷ rule out the already covered examples
        $r \leftarrow add\_head(r,\ l)$
        $R \leftarrow R \cup \{r\}$
    end while
    return $R$
end function
function MOST($E$)                    ▷ find the most popular target literal
    for $e \in E$ do
        $count[label_e] \leftarrow count[label_e] + 1$
    end for
    $label_{most} \leftarrow$ FIND_MOST(count)
    return $literal(index_{label}, =, label_{most})$
end function
function SPLIT_BY_LITERAL($E$, $l$)
    $E^+, E^- \leftarrow \emptyset, \emptyset$
    for $e \in E$ do
        if EVALUATE($e$, $l$) is true then
            $E^+ \leftarrow E^+ \cup \{e\}$
        else
            $E^- \leftarrow E^- \cup \{e\}$
        end if
    end for
    return $E^+, E^-$
end function
```

Example 2
The target is to learn rules for `habitat` using the FOLD-RM algorithm. B, E are background knowledge and training examples, respectively. There are 3 classifications: two explicit ones (land and water), and one implicit one (neither land, nor water).

```
B:  mammal(kitty).          cat(kitty).
    mammal(john).           whale(john).
    mammal(smoky).          bear(smoky).
    mammal(charlie).          dog(charlie).
    fish(nemo).               clownfish(nemo).
E:  habitat(charlie,land).    habitat(john,water).
    habitat(smoky,land).      habitat(nemo,water).
    habitat(kitty,land).
```

For the first rule, the target predicate {`habitat(X,land):- true`} is specified at line 4 in Algorithm 4 because 'land' is the majority label. The `find_best_literal` function selects literal `mammal(X)` as result and adds it to the clause r = {`habitat(X,land):- mammal(X)`} at line 17 in Algorithms 3 because it provides the most useful information among literals {`cat,whale,bear,dog,fish,clownfish`}. Then the training set rules out covered examples at line 20–21 in Algorithm 3, $E^+ = \emptyset$, E^- ={`john,nemo`}. The default learning is finished at this point because the candidate literal cannot provide any further useful information. Therefore, the `fold_rpp` function is called recursively with swapped positive and negative examples, E^+ ={`john,nemo`}, $E^- = \emptyset$, to learn exceptions. In this case, an abnormal predicate {`ab1(X):-whale(X)`} is learned and added to the previously generated clause as r = {`habitat(X,land):- mammal(X), not ab1(X)`}. And the exception rule {`ab1(X):- whale(X)`} is added to the answer set program. FOLD-RM next learns rules for target predicate {`habitat(X,water):- true`} and two rules are generated as {`habitat(X,water):- fish(X)`} and {`habitat(X,water):- whale(X)`}. The generated final answer set program is:

```
habitat(X,land):- mammal(X), not ab1(X).
habitat(X,water):- fish(X).
habitat(X,water):- whale(X).
ab1(X):- whale(X).
```

The program above is a logic program, which means rules are not mutually exclusive. For correctness, a rule should be checked only if all the earlier rules result in failure. FOLD-RM generates further rules to make the learned rules mutually exclusive. The program above is transformed as shown below.

```
habitat(X,land):- habitat_1(X).
habitat(X,water):- habitat_2(X), not habitat_1(X).
habitat(X,water):- habitat_3(X), not habitat_2(X), not habitat_1(X).
habitat_1(X):- mammal(X), not ab1(X).
habitat_2(X):- fish(X).
habitat_3(X):- whale(X).
ab1(X):- whale(X).
```

5 Experimental results

In this section, we present our experiments on standard UCI benchmarks. The XGBoost classifier is a well-known classification model and used as a baseline model in our experiments. The settings used for XGBoost classifier is kept simple without limiting its performance. MLP is another widely used classification model that can deal with generic classification tasks. However, both XGBoost model and MLP cannot take mixed type (numerical and categorical values in a row or a column) as training data *without pre-*

processing. For mixed type data, one-hot encoding—as explained in the book by Aggarwal (2018)—has been used for data preparation. For binary classification, we use accuracy, precision, recall, and F_1 score as evaluation metrics. For the multi-category classification tasks, following convention, we use accuracy, weighted average of Macro precision, weighted average of Macro recall, and weighted average of Macro F_1 score to compare models as explained by Grandini *et al.* (2020). The average numbers of generated rules are also reported for the FOLD-R++ and FOLD-RM algorithms in Tables 4, 6, and 7, some of them are not integers because of being averaged over a number of repeated experiments.

Both FOLD-R++ and FOLD-RM algorithms *do not* need any encoding for training. After specifying the numerical features, they can deal with mixed type data directly, that is, no one-hot encoding is needed. Even missing values are handled and do not need to be provided. We implemented both algorithms with Python. The hyper-parameter *ratio* is simply set as 0.5 for all the experiments. And all the learning processes have been run on a desktop with Intel i5-10400 CPU @ 2.9 GHz and 32 GB RAM. To have good performance test, we performed 10-fold cross-validation test on each dataset and average classification metrics and execution time are shown. The best performer is highlighted in boldface.

The XGBoost classifier utilizes decision tree ensemble method to build model and provides good performance. Performance comparison of FOLD-R++ and XGBoost is shown in Table 4. The FOLD-R++ algorithm is comparable to XGBoost classifier for classification, but it is more efficient in terms of execution time, especially on datasets with many unique feature values.

For multi-category classification experiments, we collected 15 datasets for comparison with XGBoost and MLP. The drug consumption dataset has many output attributes, we perform training on heroin, crack, and semer attributes. The size and label distribution of the datasets used is shown in Table 5: number of rows indicates the number of data records, while the number of columns indicates the number of features. We first compare the performance of FOLD-RM and XGBoost in Table 6. XGBoost performs much better on datasets avila and yeast, and FOLD-RM performs much better on datasets ecoli, dry-bean, eeg, and weight-lifting. After analyzing these dataset, FOLD-RM seems to perform better on more complicated datasets with mixed type values. XGBoost seems to perform better on the datasets that have limited information. However, for those datasets for which FOLD-RM has similar performance with XGBoost, FOLD-RM is more efficient in terms of execution speed. In addition, FOLD-RM is explainable/interpretable, and XGBoost is not.

The comparison with MLP is presented in Table 7. For most datasets, FOLD-RM can achieve equivalent scores, similar to the comparison with XGBoost, FOLD-RM performs much better on datasets ecoli, dry-bean, eeg, and weight-lifting, while MLP performs much better on datasets avila and yeast. MLP takes much more time for training than XGBoost because of its algorithmic complexity. Like the XGBoost classifier, for complex datasets with mixed values, MLP also suffers from pre-processing complications such as having to use one-hot encoding.

RIPPER algorithm by Cohen (1995) is a popular rule induction algorithm that generates conjunctive normal form (CNF) formulas. Eight datasets have been used for comparison between RIPPER and FOLD-RM. We did not find the RIPPER algorithm

https://doi.org/10.1017/S1471068422000205 Published online by Cambridge University Press

Table 4. *Comparison of XGBoost and FOLD-R++ on various Datasets*

DataSet			XGBoost.Classifier					FOLD-R++					
Name	#Rows	#Cols	Acc	Prec	Rec	F1	T(ms)	Acc	Prec	Rec	F1	T(ms)	#Rules
acute	120	7	**1**	1	**1**	**1**	35	0.99	1	0.99	0.99	**1.6**	2.6
autism	704	18	**0.97**	**0.98**	**0.98**	0.97	76	0.95	0.96	0.97	0.97	**47**	24.3
breast-w	699	10	0.95	0.97	0.96	0.96	78	**0.96**	0.97	0.96	**0.97**	**24**	10.2
cars	1728	7	**1**	1	**1**	**1**	77	0.98	1	0.97	0.98	**38**	12.2
credit-a	690	16	**0.85**	0.83	**0.83**	0.83	368	0.84	**0.92**	0.79	**0.84**	**66**	10.0
ecoli	336	9	0.76	0.76	0.62	0.68	165	**0.96**	**0.95**	**0.94**	**0.95**	**19**	11.4
heart	270	14	**0.80**	**0.81**	0.83	0.81	112	0.79	0.79	0.83	0.81	**24**	11.7
ionosphere	351	35	0.88	0.86	**0.96**	0.90	1,126	**0.92**	**0.93**	0.94	**0.93**	**214**	12.0
kidney	400	25	0.98	0.98	0.98	0.98	126	**0.99**	**1**	0.98	**0.99**	**19**	5.0
kr vs. kp	3196	37	0.99	0.99	0.99	0.99	**210**	0.99	0.99	0.99	0.99	223	18.4
mushroom	8124	23	1	1	1	1	378	1	1	1	1	**314**	8.0
voting	435	17	0.95	0.94	**0.95**	0.94	49	0.95	0.94	0.94	0.94	**17**	10.5
adult	32561	15	**0.86**	**0.88**	0.94	**0.91**	274,655	0.84	0.86	**0.95**	0.90	**2,546**	16.7
rain in aus	145460	24	**0.83**	0.84	**0.95**	**0.89**	285,307	0.78	**0.87**	0.84	0.85	**21,868**	40.5

Table 5. *The size and label distribution of UCI datasets*

Dataset	#Rows	#Cols	Distribution
anneal	898	39	‘3’: 684, ‘U’: 40, ‘1’: 8, ‘5’: 67, ‘2’: 99
avila	20867	11	‘A’:8572,‘F’:3923,‘H’:1039,‘E’:2190,‘I’:1663,‘Y’:533 ‘D’:705,‘X’:1044,‘G’:893,‘W’:89,‘C’:206,‘B’:10
coli	336	9	‘cp’:143,‘im’:77,‘imS’:2,‘imL’:2,‘imU’:35,‘om’:20,‘omL’:5, ‘pp’:52
drug-heroin	1885	13	‘CL0’:1605,‘CL1’:68,‘CL2’:94,‘CL3’:65,‘CL5’:16,‘CL6’:13, ‘CL4’:24
drug-crack	1885	13	‘CL0’:1627,‘CL1’:67,‘CL2’:112,‘CL3’:59,‘CL5’:9,‘CL4’:9, ‘CL6’:2
drug-semer	1885	13	‘CL0’: 1877, ‘CL2’: 3, ‘CL3’: 2, ‘CL4’: 1, ‘CL1’:2
dry-bean	13611	17	‘SEKER’: 2027, ‘BARBUNYA’: 1322, ‘BOMBAY’: 522 ‘CALI’: 1630, ‘HOROZ’: 1928, ‘SIRA’: 2636, ‘DERMASON’: 3546
eeg	14980	15	‘0’: 8257, ‘1’: 6723
intention	12330	18	‘FALSE’: 10422, ‘TRUE’: 1908
nursery	12960	9	‘recommend’: 2, ‘priority’: 4266 ‘not_recom’: 4320, ‘very_recom’: 328, ‘spec_prior’: 4044
pageblocks	5473	11	‘1’: 4913, ‘2’: 329, ‘4’: 88, ‘5’: 115, ‘3’: 28
parkinson	756	754	‘1’: 564, ‘0’: 192
pendigits	10992	17	‘8’: 1055, ‘2’: 1144, ‘1’: 1143, ‘4’: 1144, ‘6’: 1056 ‘0’: 1143, ‘5’: 1055, ‘9’: 1055, ‘7’: 1142, ‘3’: 1055
wine	178	14	‘1’: 59, ‘2’: 71, ‘3’: 48
weight-lift	4024	155	‘E’: 1370, ‘A’: 1365, ‘D’: 276, ‘B’: 901, ‘C’: 112
yeast	1484	10	‘MIT’: 244, ‘NUC’: 424, ‘CYT’: 463, ‘ME1’: 44, ‘EXC’: 35, ‘ME2’: 51 ‘ME3’: 163, ‘VAC’: 30, ‘POX’: 20, ‘ERL’: 5, ‘0.18’: 2, ‘0.16’: 2, ‘0.37’: 1
wall-robot	5456	25	‘Slight-Right-Turn’: 826, ‘Sharp-Right-Turn’: 2097 ‘Move-Forward’: 2205, ‘Slight-Left-Turn’: 328
flags	194	10	‘2’: 36, ‘6’: 15, ‘1’: 60, ‘0’: 40, ‘5’: 27, ‘3’: 8, ‘4’: 4, ‘7’: 4
glass	214	10	‘1’: 70, ‘2’: 76, ‘3’: 17, ‘5’: 13, ‘6’: 9, ‘7’: 29
optidigits	3823	65	‘0’: 376, ‘7’: 387, ‘4’: 387, ‘6’: 377, ‘2’: 380 ‘5’: 376, ‘8’: 380, ‘1’: 389, ‘9’: 382, ‘3’: 389
shuttle	58000	10	‘2’: 50, ‘4’: 8903, ‘1’: 45586, ‘5’: 3267, ‘3’: 171, ‘7’: 13, ‘6’: 10

implementation with multiclass classification. Therefore, we have collected the accuracy data reported by Asadi and Shahrabi (2016) and performed the same experiment with the same datasets with the FOLD-RM algorithm. Two-thirds of the dataset was used for training by Asadi and Shahrabi (2016) and the remaining one-third used as the test set. We follow the same convention. For each dataset, this process was repeated 50 times. The average of accuracy is shown in Table 8. Both algorithms have similar accuracy on most datasets, though FOLD-RM outperforms on nursery dataset. Ripper is explainable, as it outputs CNF formulas. However, the CNF formulae generated tend to have large number of literals. In contrast, FOLD-RM rules are succinct due to use of NAF and they have an operational semantics (that aligns with how humans reason) by virtue of being a normal logic program.

Table 6. *Comparison of FOLD-RM and XGBoost on UCI Datasets*

	FOLD-RM						XGBoost				
Dataset	Acc	Prec	Rec	F1	Rules	T(ms)	Acc	Prec	Rec	F1	T(ms)
anneal	0.99	**1**	0.99	0.99	17.9	**63**	0.99	0.99	0.99	0.99	295
avila	0.33	0.49	0.33	0.39	33.6	**3,540**	**1**	**1**	**1**	**1**	4,897
ecoli	**0.80**	**0.82**	**0.80**	**0.80**	42.3	**41**	0.42	0.79	0.42	0.50	806
drug-heroin	0.84	0.74	0.84	0.78	12.0	**136**	0.84	**0.77**	0.84	**0.79**	1,266
drug-crack	0.85	0.75	0.85	0.80	17.6	**145**	0.85	**0.76**	0.85	**0.81**	1,116
drug-semer	0.99	0.99	0.99	0.99	10.3	**43**	**1**	0.99	**1**	0.99	393
dry-bean	**0.91**	**0.91**	**0.91**	**0.91**	185.7	13,415	0.29	0.87	0.29	0.37	**3,458**
eeg	**0.78**	**0.78**	**0.78**	**0.77**	164.5	3,914	0.50	0.70	0.50	0.54	**340**
intention	0.90	0.89	0.90	**0.90**	78.3	**1,621**	0.90	0.89	0.90	0.89	114,161
nursery	**0.97**	**0.97**	**0.97**	**0.96**	59.8	**643**	0.88	0.93	0.88	0.89	24,100
pageblocks	**0.97**	**0.97**	**0.97**	**0.96**	72.3	**929**	0.95	0.94	0.95	0.94	81,416
parkinson	0.81	0.80	0.81	0.79	15.9	8,503	**0.84**	**0.84**	**0.84**	**0.83**	**527**
pendigits	**0.96**	**0.96**	**0.96**	**0.96**	219.2	**2,447**	0.91	0.92	0.91	0.91	54,102
wine	**0.94**	0.97	**0.94**	0.95	7.6	**17**	0.93	**1**	0.93	**0.96**	49
weight-lift	**1**	**1**	**1**	**1**	14.0	**1,879**	0.51	0.81	0.51	0.57	224,140
yeast	0.08	0.15	0.08	0.10	8.7	**146**	**0.45**	**0.5**	**0.45**	**0.45**	8,629
wall-robot	0.99	0.99	0.99	0.99	30.5	2,402	0.99	0.99	0.99	0.99	**403**

Table 7. *Comparison of FOLD-RM and MLP on UCI Datasets*

	FOLD-RM						MLP				
Dataset	Acc	Prec	Rec	F1	Rules	T(ms)	Acc	Prec	Rec	F1	T(ms)
anneal	0.99	**1**	0.99	0.99	17.9	**63**	0.99	0.99	0.99	0.99	462
avila	0.33	0.49	0.33	0.39	33.6	**3,540**	**0.90**	**0.90**	**0.90**	**0.90**	73,610
ecoli	**0.80**	0.82	**0.80**	**0.80**	42.3	**41**	0.52	**0.91**	0.52	0.61	411
drug-heroin	**0.84**	0.74	**0.84**	0.78	12.0	**136**	0.82	**0.77**	0.82	**0.79**	1,093
drug-crack	**0.85**	0.75	**0.85**	0.80	17.6	**145**	0.84	0.77	0.84	0.80	1,061
drug-semer	0.99	0.99	0.99	0.99	10.3	**43**	**1**	0.99	**1**	0.99	518
dry-bean	**0.91**	0.91	**0.91**	**0.91**	185.7	13,415	0.57	**0.92**	0.57	0.66	**11,292**
eeg	**0.78**	**0.78**	**0.78**	**0.77**	164.5	**3,914**	0.49	0.68	0.49	0.54	5,946
intention	**0.90**	**0.89**	**0.90**	**0.90**	78.3	**1,621**	0.84	0.76	0.84	0.78	218,087
nursery	**0.97**	**0.97**	**0.97**	**0.96**	59.8	**643**	0.91	0.94	0.91	0.91	943
pageblocks	**0.97**	**0.97**	**0.97**	**0.96**	72.3	**929**	0.93	0.91	0.93	0.92	6,452
parkinson	0.81	0.80	0.81	0.79	15.9	8,503	**0.82**	**0.82**	**0.82**	**0.81**	**1,416**
pendigits	0.96	0.96	0.96	0.96	219.2	**2,447**	**0.99**	**0.99**	**0.99**	**0.99**	6,732
wine	0.94	0.97	0.94	0.95	7.6	**17**	**0.97**	**1**	**0.97**	**0.98**	189
weight-lift	**1**	**1**	**1**	**1**	14.0	**1,879**	0.54	0.89	0.54	0.58	52,643
yeast	0.08	0.15	0.08	0.10	8.7	**146**	**0.41**	**0.49**	**0.41**	**0.38**	3,750
wall-robot	**0.99**	**0.99**	**0.99**	**0.99**	30.5	**2,402**	0.88	0.88	0.88	0.88	8,141

Table 8. *Comparison of RIPPER and FOLD-RM on UCI Datasets*

Dataset	RIPPER Acc	FOLD-RM Acc	Dataset	RIPPER Acc	FOLD-RM Acc
ecoli	0.80	0.80	flags	**0.61**	0.58
glass	0.63	0.63	nursery	0.72	**0.96**
optidigits	0.90	0.90	pageblocks	**0.97**	0.96
pendigits	0.95	0.95	shuttle	0.99	**1**

6 Prediction and justification

The FOLD-RM algorithm generates rules that can be interpreted by the human user to understand the patterns and correlations that are implicit in the table data. These rules can also be used to make prediction given new data input. Thus FOLD-RM serves as a machine learning algorithm in its own right. However, making good predictions is not enough for critical tasks such as disease diagnosis and loan approval. FOLD-RM comes with a built-in prediction and justification facility. We illustrate this justification facility via an example.

Example 3
The "annealing" UCI dataset is a multi-category classification task which contains 798 training examples and 100 test examples and their classes based on features such as steel, carbon, hardness, condition, strength, etc. FOLD-RM generates the following answer set program with 20 rules for 5 classes, which is pretty concise and precise:

```
classes(X,'3') :- not surface_quality(X,'?'), not ab1(X),
                  not ab2(X), not ab3(X), not ab4(X).
classes(X,'2') :- condition(X,'s'), not ab5(X).
classes(X,'3') :- not carbon(X,'00'), not ab6(X).
classes(X,'5') :- family(X,'tn').
classes(X,'u') :- steel(X,'a'), not ab7(X).
classes(X,'2') :- thick(X,N32), N32>0.8, not ab8(X),
                not ab9(X), not ab10(X).
classes(X,'3') :- not steel(X,'s'), not ab11(X), not ab6(X).
classes(X,'1') :- family(X,'?').
classes(X,'1') :- family(X,'zs').
ab1(X) :- hardness(X,'85').
ab2(X) :- strength(X,'600').            ab3(X) :- carbon(X,'10').
ab4(X) :- hardness(X,'80'), cbond(X,'?').
ab5(X) :- not steel(X,'r'), not enamelability(X,'2').
ab6(X) :- steel(X,'a').                 ab7(X) :- carbon(X,'03').
ab8(X) :- steel(X,'r').                 ab9(X) :- steel(X,'s').
ab10(X) :- not temper_rolling(X,'?').   ab11(X) :- not family(X,'?').
```

The above generated rule set achieves 0.99 accuracy, 0.99 weighted Macro precision, 0.99 weighted Macro recall, and 0.99 weighed Macro F1 score. The justification tree generated by the FOLD-RM system for the 8^{th} test example is shown below:

```
Proof Tree for example number 8 :
the value of classes is 2 DOES HOLD because
    the value of condition is 's' which should equal 's' (DOES HOLD)
    exception ab5 DOES NOT HOLD because
        the value of steel is 'r' which should not equal 'r' (DOES NOT HOLD)
        the value of enamelability is '?' which should not equal '2' (DOES HOLD)
{'condition: S', 'enamelability: ?', 'steel: R'}
```

This justification tree is also shown in another format: by showing which rules were involved in the proof/justification. For each call in each rule that was invoked, FOLD-RM shows whether it is true ([T]) or false ([F]). The head of each applicable rule is similarly annotated. We illustrate this for the 8th test example:

```
[F]ab5(X) :- not [T]steel(X,'r'), not [F]enamelability(X,'2').
[T]classes(X,'2') :- [T]condition(X,'s'), not [F]ab5(X).
{'condition: S', 'enamelability: ?', 'steel: R'}
```

7 Conclusions and related work

In this paper we presented FOLD-RM, an efficient and highly scalable algorithm for multi-category classification tasks. FOLD-RM can generate explainable answer set programs and human-friendly justification for predictions. Our algorithm does not need any encoding (such as one-hot encoding) for data preparation. Compared to the well-known classification models like XGBoost and MLP, our new algorithm has similar performance in terms of accuracy, weighted macro precision, weighted macro recall, and weighted macro F_1 score. However, our new approach is much more efficient and interpretable than these other approaches. It is remarkable that an ILP system is comparable in accuracy to state-of-the-art traditional machine learning systems.

ALEPH by Srinivasan (2001) is a well-known ILP algorithm that employs bottom-up approach to induce rules for non-numerical data. Also, no automatic method is available for the specialization process. A tree-ensemble based rule extraction algorithm is proposed by Takemura and Inoue (2021), its performance relies on trained tree-ensemble model. It may also suffer from scalability issue because its running time is exponential in the number of valid rules.

In practice, statistical machine learning models show good performance for classification. Extracting rules from statistical models is also a long-standing research topic. Rule extraction algorithms are of two kinds: 1) pedagogical (learning rules from black box models without looking into internal structures), such as, TREPAN by Craven and Shavlik (1995), which learns decision trees from neural networks, 2) decompositional (learning rules by analyzing the models inside out) such as SVM+Prototypes by Nuez *et al.* (2006), which employs clustering algorithm to extract rules from SVM classifiers by utilizing support vectors. RuleFit by Friedman and Popescu is another rule extraction algorithm that learns sparse linear models with original feature decision rules from shallow tree-ensemble model for both classification and regression tasks. However, its interpretability decreases when too many decision rules have been generated. Also, simpler approaches that are a combination of statistical method with ILP have been extensively explored. The kFOIL system by Landwehr *et al.* (2006) incrementally learns kernel for SVM FOIL style rule induction. The nFOIL system by Landwehr *et al.* (2005) is an integration of Naive Bayes model and FOIL. TILDE by Blockeel and De Raedt (1998) is another top-down rule induction algorithm based on C4.5 decision tree, it can achieve similar performance with decision tree. However, it would suffer from scalability issue when there are too many unique numerical values in the dataset. For most datasets we experimented with, the number of leaf nodes in the trained C4.5 decision tree is much more than the number of rules that FOLD-R++/FOLD-RM generate. The FOLD-RM algorithm outperforms the above methods in efficiency and scalability due to (i) its use of learning defaults, exceptions to defaults, exceptions to exceptions, and so on (ii) its top-down nature, and (iii) its use of improved method (prefix sum) for heuristic calculation.

Acknowledgment

Authors acknowledge support from NSF grants IIS 1718945, IIS 1910131, IIP 1916206, US DoD, Atos Corp and Amazon Corp. We thank our colleagues Joaquin Arias, Parth Padalkar, Kinjal Basu, Sarat Chandra Varanasi, Elmer Salzar, Fang Li, Serdar Erbatur, and Doug DeGroot for discussions and help.

References

Aggarwal, C. C. 2018. *Neural Networks and Deep Learning - A Textbook*. Springer.

Arias, J., Carro, M., Chen, Z. and Gupta, G. 2020. Justifications for goal-directed constraint answer set programming. *Electronic Proceedings in Theoretical Computer Science*, *325*, 59–72.

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.

Asadi, S. and Shahrabi, J. 2016. Ripmc: Ripper for multiclass classification. *Neurocomputing*, *191*, 19–33.

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

Bishop, C. M. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

Blockeel, H. and De Raedt, L. 1998. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, *101*, 1, 285–297.

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

Cohen, W. W. 1995. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on International Conference on Machine Learning* 1995, ICML95, 115–123, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

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

Cropper, A. and Dumancic, S. 2020. Inductive logic programming at 30: A new introduction. URL: https://arxiv.org/abs/2008.07912

Dua, D. and Graff, C. 2017. UCI machine learning repository.

Gelfond, M. and Kahl, Y. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press.

Grandini, M., Bagli, E. and Visani, G. 2020. Metrics for multi-class classification: an overview. arXiv 2008.05756.

Gunning, D. 2015. Explainable artificial intelligence (xai). URL: https://www.darpa.mil/program/explainable-artificial-intelligence [Accessed on June 2018].

Landwehr, N., Kersting, K. and Raedt, L. D. 2005. nFOIL: Integrating naïve bayes and FOIL. In *Proc. AAAI* 2005, 795–800.

Landwehr, N., Passerini, A., Raedt, L. D. and Frasconi, P. 2006. kFOIL: Learning simple relational kernels. In *Proc. AAAI* 2006, 389–394.

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

Muggleton, S., de Raedt, L., Poole, D., Bratko, I., Flach, P., Inoue, K. and Srinivasan, A. 2012. ILP turns 20. *Machine Learning*, *86*, 1, 3–23.

Nuez, H., Angulo, C. and Catal, A. 2006. Rule-based learning systems for support vector machines. *Neural Processing Letters*, *24*, 1–18.

Plotkin, G. D. 1971. A further note on inductive generalization. In *Machine Intelligence* 1971, volume 6. Edinburgh University Press, 101–124.

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

Reiter, R. 1980. A logic for default reasoning. *Artificial Intelligence*, *13*, 1-2, 81–132.

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

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

Shakerin, F. and Gupta, G. 2018. Heuristic based induction of answer set programs, from default theories to combinatorial problems. In *28th International Conference on Inductive Logic Programming (ILP Up and Coming Papers)* 2018, volume 2206, pp. 36–51.

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

Srinivasan, A. 2001. The aleph manual. URL: http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/

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.

Wang, H. and Gupta, G. 2022. FOLD-R++: A toolset for automated inductive learning of default theories from mixed data. In *Proceedings The First International Workshop on Combining Learning and Reasoning: Programming Languages, Formalisms, and Representations (CLeaR)*.

Zeng, Q., Patel, J. M. and Page, D. 2014. Quickfoil: Scalable inductive logic programming. *Proceedings of the VLDB Endowment*, *8*, 3, 197–208.