Generalizing Conjunctive and Disjunctive Rule Learning to Learning m-of-n Concepts

Florian Beck[1], Johannes Fürnkranz[1] and Van Quoc Phuong Huynh[1]

[1]Johannes Kepler University Linz, Department of Computer Science, Institute for Application-oriented Knowledge Processing (FAW), Linz, Austria

Abstract

Most rule learning algorithms learn rule concepts as conjunctions and disjunct them afterwards to rule sets, a few others swap the order of conjunction and disjunction so that rule concepts are learned as disjunctions. Depending on the domain, both approaches can have advantages or disadvantages in comparison to its counterpart.

Instead of learning rule concepts only as conjunctions or only as disjunctions, one can also flexibly choose between these two representations. One way to do so is by using m-of-n concepts where m of conditions must be true in order for the expression to be true. This not only covers the two extreme cases where all conditions must be true (n-of-n, conjunctions) or any of them must be true (1-of-n, disjunctions) but also a smooth transition for other values of m, analogous to a customizable activation threshold in neural networks.

In this paper, we discuss possibilities how to efficiently learn m-of-n rules using similar generalization and specialization operations as for conjunctions or disjunctions. Furthermore, we adjust the state-of-the-art rule learning algorithm LORD to learn m-of-n concepts instead of plain conjunctions and present an evaluation of the technique on artificial and real-world data sets.

Keywords

rule learning, constructive induction, m-of-n concepts

1. Introduction

While most rule learning algorithms stick to learning flat concepts as logical combinations of features, many recent approaches — most notably neural networks — use threshold concepts instead. In threshold concepts, the contained features are associated with different weights, and not necessarily all of them have to be present in order to pass the threshold. This leads to more flexible representations than in rules, where all features contribute equally and a concept is fulfilled if all features are present (conjunctive rule) or any of the features is present (disjunctive rule).

A smooth transition between the two approaches is provided by m-of-n concepts: Of a given set of n features at least m have to be present to fulfill the concept. One of the most well-known usages of m-of-n concepts in symbolic approaches was in the domain of decision trees, namely by ID-2-of-3, which integrates m-of-n discriminators and could outperform standard decision tree induction in some domains while also providing smaller trees [1].

In the area of rule learning, m-of-n concepts are mostly used when extracting concepts from neural networks and transforming weights and node activations into rules that are easier to understand [2][3][4]. Moreover, these extracted rules often generalize better to examples not seen during training than rules produced by "all-symbolic" rule refinement algorithms [2]. Additionally, there has also been work on how these extracted rules can be unified to obtain a smaller and potentially easier understandable rule set [5].

One of the few approaches that directly integrate m-of-n concepts in a rule learner is Neither-MofN [6]. In comparison to its counterpart algorithm that does not use m-of-n concepts, it was able to generate less complex theories because it could directly modify threshold values rather than create new rules. By adding one more operator to the generalization and specialization processes, Neither-MofN was able to accurately revise a theory known to be difficult for symbolic systems, without having to sacrifice the efficiency of a symbolic approach [6].

In this paper, we further investigate into this incremental rule learning technique, using m-of-n concepts in the form of Boolean expressions within a state-of-the-art rule learner. Our goal is to analyze the performance of the learner using m-of-n-concepts in comparison to conventional rule learners, with the long-term perspective of using m-of-n concepts as modules in more sophisticated, deeper rule learners.

The remainder of the paper is organized as follows: Section 2 describes how m-of-n concept can be represented and learned. Based on this, in Section 3, we propose a

ITAT'23: Information technologies – Applications and Theory, September 22–26, 2023, Tatranské Matliare, Slovakia

fbeck@faw.jku.at (F. Beck); juffi@faw.jku.at (J. Fürnkranz); vqphuynh@faw.jku.at (V. Q. P. Huynh)
0000-0003-3183-2953 (F. Beck); 0000-0002-1207-0159 (J. Fürnkranz)

novel rule learning algorithm using a dynamic programming approach to compute m-of-n concepts efficiently, and test it in Section 4. Finally, the results are concluded in Section 5, and future ideas are discussed in Section 6.

2. M-of-n concepts

M-of-n concepts, also known as criteria tables, are the simplest form of threshold descriptions, without any feature weights involved [7, Chapter 3]. They consist of a set of n Boolean features and a threshold m between 1 and n. If for a given example m of the n features are true, this example is a positive instance of the concept. M-of-n concepts can be written as linear combinations, e.g. $a + b + c \geq 2$, or in the form 2 of $[a, b, c]$. In this paper, we will use the latter notation.

Learning m-of-n concepts Similar to conventional rules, m-of-n concepts can be learned by starting with an arbitrary feature subset (e.g. $[a, b, c]$) and a threshold m between 1 and n (e.g. 2). Afterwards, this concept can be generalized and specialized by adding or removing features or by adjusting the threshold. Typical incremental algorithms (e.g. [1][7, Chapter 3]) use the following two generalization operations to learn m-of-n concepts:

- add a feature,
  e.g. 2 of $[a, b, c] \rightarrow$ 2 of $[a, b, c, d]$
- remove a feature and decrease m,
  e.g. 2 of $[a, b, c] \rightarrow$ 1 of $[a, b]$

Accordingly, the specialization operators are:

- remove a feature,
  e.g. 2 of $[a, b, c] \rightarrow$ 2 of $[a, b]$
- add a feature and increase m,
  e.g. 2 of $[a, b, c] \rightarrow$ 3 of $[a, b, c, d]$

These generalization and specialization operations are minimal; i.e. all other operations can be achieved by chaining these operations together. For example, the generalization of decreasing the threshold m by 1 just combines both minimal generalization operations:

- 2 of $[a, b, c] \rightarrow$ 2 of $[a, b, c, d] \rightarrow$ 1 of $[a, b, c]$

M-of-n concepts as Boolean expressions. M-of-n concepts can also be formulated as Boolean expressions. E.g., they can be expressed in disjunctive normal form (DNF) in a straightforward way, by listing all $\binom{n}{m}$ possibilities to build a conjunction of m of n features, and disjunct all these conjunctions. The length of this DNF is then $\binom{n}{m} * m$.

An even shorter form with less literals can be achieved if the representation is not limited to two layers. Using the distributive law, e.g. (a ∧ b) ∨ (a ∧ c) ∨ (b ∧ c) can be rewritten as (a ∧ b) ∨ [(a ∨ b) ∧ c], which requires one literal and one conjunction less. However, even more important than reducing the number of literals and conjunction or disjunction operations is the fact that these representations can reuse representations with smaller m and n. Hereby, the general idea is to distinguish between two cases, how a given m-of-n concept can be fulfilled: (a) Either among the first $n - 1$ features already m are true or (b) among the first $n - 1$ features $m - 1$ are true, and the last feature is true as well. Thus, this representation can be generated by the following recurrence system:

Table 1
Dynamic programming approach to compute m-of-n concepts as Boolean expressions. For the concepts 2-of-3, 2-of-4 and 3-of-4, at the top the representation in DNF and at the bottom the shorter representation computed by dynamic programming is shown.

n / m	1	2	3	4
1	a	a ∨ b	a ∨ b ∨ c	a ∨ b ∨ c ∨ d
2	—	a ∧ b	(a ∧ b) ∨ (a ∧ c) ∨ (b ∧ c) = (a ∧ b) ∨ [(a ∨ b) ∧ c]	(a ∧ b) ∨ (a ∧ c) ∨ (b ∧ c) ∨ (a ∧ d) ∨ (b ∧ d) ∨ (c ∧ d) = (a ∧ b) ∨ [(a ∨ b) ∧ c] ∨ [(a ∨ b ∨ c) ∧ d]
3	—	—	a ∧ b ∧ c	(a ∧ b ∧ c) ∨ (a ∧ b ∧ d) ∨ (a ∧ c ∧ d) ∨ (b ∧ c ∧ d) = a ∧ b ∧ c ∨ {(a ∧ b) ∨ [(a ∨ b) ∧ c]} ∧ d
4	—	—	—	a ∧ b ∧ c ∧ d

$$expr(m, n) = \begin{cases} \text{true} & \text{if m=0} \\ \text{false} & \text{if m>n} \\ expr(m, n-1) \vee & \\ [expr(m-1, n-1) \wedge x_n] & \text{else} \end{cases} \tag{1}$$

Note that there are two ways how the recurrence terminates: If no more feature needs to be fulfilled, i.e. if $m = 0$, this m-of-n subconcept is always true. Analogously, if more features need to be fulfilled than exist in a concept, i.e. if $m > n$, this m-of-n subconcept is always false.

If applied recursively, most "subexpressions" will appear multiple times in the recursion — the subproblems to be solved are nested. E.g., $expr(m - 1, n - 2)$ appears as the first term in the recurrent formula for $expr(m - 1, n - 1)$ and also as the second term in the recurrent formula for $expr(m, n - 1)$. We can make use of dynamic programming to cope with these reoccurring subproblems.

Table 1 gives an overview how the dynamic programming approach can be used to generate the shorter Boolean representation for the values $m, n \in [1..4]$ and sample features $x_1 = a, x_2 = b, x_3 = c, x_4 = d$ according to recurrence system 1. Note that for $m = 1$ the expression is a flat disjunction and for $m = n$ a flat conjunction. These are the two border cases before the most general concept "all true" ($m = 0$) or the most specific concept "all false" ($m > n$) are reached.

In the remaining three cases, first the (longer) DNF and below the recursively computed expression are listed. For example, the recursively computed expression for $m = 2, n = 4$ consists of the expression for $m = 2, n = 3$ (5 literals), and disjuncts this with the conjunction of the expression for $m = 1, n = 3$ (3 literals) and $x_4 = d$ (1 literal).

3. Algorithm

To evaluate the presented approach of recursively generating m-of-n concepts, we will adjust Lord [8], a novel rule learner developed in our group. Regularly, Lord learns one conjunctive rule for each training example and groups them (including some filtering) to a DNF rule set which is used for classification. In its extended version, Lord should be capable to learn an arbitrary m-of-n concept per training example instead.

Data structures in Lord. Lord builds upon data structures that are well-known in association rule learning, namely PPC-Trees[1] and n-Lists. The main idea is that only a single pass through the training data is required during the learning phase. This pass is used to create the PPC-Tree where each path corresponds to a (unique) example. Afterwards, n-Lists can be used to determine which parts of a tree are affected by a given Boolean expression — and therefor how many positive and negative examples are covered for a learned rule. N-lists can be combined particularly efficiently by conjunctions but can be combined by disjunctions as well. Detailed information about these data structures are given in [9], and about their application in the Lord algorithm in [8].

Greedy rule search. Lord greedily learns one conjunctive rule for each training example. It starts with an empty rule body and specializes it by greedily adding one feature at a time to it, always picking the feature with the maximum gain w.r.t. a given rule heuristic until no improvement is possible. The set of features is limited by the concerned example to ensure that it remains covered with any tested specialization. In a second step, Lord tries to prune these rules by repeatedly removing one feature contained in the body as long as it further improves the heuristic.

[1]<pre, post>-code-Trees; the code is determined for each node after a preorder and postorder traversal

Algorithm 1: `search_best_greedy_rule()`

```
   Input: example, metric, all_features, n_lists
   Output: concept
1  best_rule.body $\leftarrow$ [];
2  best_rule.heuristic $\leftarrow$ $-\infty$;
3  remaining_features $\leftarrow$ example.features;
4  while true do
5      m $\leftarrow$ best_rule.m;
6      new_rule, chosen_feature $\leftarrow$
         modify_rule(best_rule, remaining_features,
         example.class, metric, n_lists, m);
7      if new_rule = null then
8          break;
9      end
10     if chosen_feature $\neq$ null then
11         remaining_features.remove(chosen_feature);
12     end
13     best_rule $\leftarrow$ new_rule;
14 end
15 remaining_features $\leftarrow$ features $\setminus$ best_rule.body;
16 while true do
17     m $\leftarrow$ best_rule.m - 1;
18     new_rule, chosen_feature $\leftarrow$
         modify_rule(best_rule, remaining_features,
         example.class, metric, n_lists, m);
19     if new_rule = null then
20         break;
21     end
22     if chosen_feature $\neq$ null then
23         remaining_features.remove(chosen_feature);
24     end
25     best_rule $\leftarrow$ new_rule;
26 end
27 return best_rule;
```

For the m-of-n version of Lord, we use the same greedy search approach. We start with an empty rule body and $m = 0$ and specialize it by either adding a feature and increasing m or by removing a feature. Afterwards, by either removing a feature and decreasing m or by adding a feature, the rule can be generalized. Thus, while the order of specialization and generalization remains the same as in regular Lord, the rule body can be grown and pruned in both phases of the algorithm.

Algorithm 1 shows the mentioned approach in pseudocode. Lines 1-2 create an empty new "0-of-[]" concept with the worst heuristic, which is afterwards first specialized (lines 3-14) and then generalized (lines 15-26). Note that there are only two small differences between these two blocks: Firstly, during the specialization, only the features of the training example are considered while during the generalization all features (except those already contained) are considered. Secondly, the two phases call

Algorithm 2: `modify_rule()`

```
   Input: best_rule, remaining_features, example.class,
          metric, n_lists, m
   Output: new_rule, chosen_feature
1  chosen_feature ← null;
2  n_list_class ← n_lists.get("1 of " + [example.class]);
3  for feature ∈ remaining_features do
4      ext_body ← best_rule.body ∪ feature;
5      n_list ← get_n_list(n_lists, ext_body, m+1);
6      n_p ← n_list.support;
7      p ← conj(n_list, n_list_class).support;
8      n ← n_p - p;
9      heuristic ← metric.evaluate(p, n);
10     if best_rule.heuristic < heuristic or
        best_rule.heuristic = heuristic and best_rule.p < p
        then
11         best_rule ← Rule(ext_body, example.class,
            m+1, p, n, heuristic);
12         chosen_feature ← feature;
13     end
14 end
15 if best_rule.body.length > 1 then
16     for feature ∈ best_rule.body do
17         prun_body ← best_rule.body \ feature;
18         n_list ← get_n_list(n_lists, prun_body, m);
19         n_p ← n_list.support;
20         p ← conj(n_list, n_list_class).support;
21         n ← n_p - p;
22         heuristic ← metric.evaluate(p, n);
23         if best_rule.heuristic < heuristic or
            best_rule.heuristic = heuristic and best_rule.p
            < p then
24             best_rule ← Rule(prun_body,
                example.class, m, p, n, heuristic);
25         end
26     end
27 end
28 return best_rule, chosen_feature;
```

the method `modify_rule` with a different parameter m set in lines 5 and 17 respectively.

Dynamic programming of n-lists. Algorithm 2 shows the starting point for the top-down dynamic programming approach to evaluate m-of-n concepts efficiently. Lines 3-14 show the procedure for growing the rule, lines 15-27 for pruning the rule. The decisive factor for whether these operations are generalizations or specializations is parameter m, which is passed to the n-List computation method `get_n_list` in lines 5 and 18.

After the n-List is computed or retrieved, both its support (lines 6 and 19) and the support of its conjunction with the predictive class (lines 7 and 20) is used to determine the number of positive (p) and negative covered examples (n). These values are used in the metric function, which also has access to the total number of positive (P) and negative examples (N) to determine a heuristic for the rule. The rule with the best heuristic — if tied, the one with more positive examples covered — is returned, optionally with the feature added in case the rule was extended.

Algorithm 3: `get_n_list()`

```
   Input: n_lists, body, m
   Output: new_rule, chosen_feature
1  n_list ← n_lists.get(m + " of " + body);
2  if n_list ≠ null then
3      return n_list;
4  end
5  n_list ← n_lists.get("1 of " + [body[body.length-1]]);
6  if m > 1 then
7      n_list ← conj(n_list, get_n_list(n_lists,
        body[0..body.length-2], m-1));
8  end
9  if body.length > m then
10     n_list_2 ← get_n_list(n_lists,
        body[0..body.length-2], m);
11 end
12 if n_list.support > 0 and n_list_2.support > 0 then
13     n_list ← disj(n_list, n_list_2);
14 else if n_list.support = 0 then
15     n_list ← n_list_2;
16 n_lists.put(m + " of " + body, n_list);
17 return n_list;
```

Finally, Algorithm 3 demonstrates how n-Lists are fetched from storage if available and otherwise computed recursively. Parameter `n_lists` is prefilled with n-Lists for every single feature (including class features), these will be retrieved immediately (line 5, also line 2 in algorithm 2). If no border cases or empty n-Lists occurz, the recurrent formula is applied so that recursive n-Lists are computed (lines 7 and 10) and combined by disjunction (line 13). All intermediate n-Lists are stored for future calls before the final n-List is returned.

Evaluation. While LORD generates a filtered rule tree in the classifier where the best covering rules for a test example can efficiently be extracted from, this is unfortunately not possible for arbitrary m-of-n concepts that are not limited to conjunctions. Instead, all rules learned during the training phase are sorted by their heuristic and iterated from best to worst whether they cover a given test example. Because both the features in the example and also the features in the m-of-n rules are sorted, these two feature lists can be iterated parallel while counting the matching features. The coverage check can stop early if the value m is already reached (return predictive class of m-of-n rule) or if m can not be reached anymore (continue with next rule).

```
[(c1=4),(c3=4),(c2=4),(c4=4)]->(class=1)  (p=884.0, n=0.0, m=2/4, heuristic_value=0.9981981688891336)
[(c1=1),(c2=1),(c3=1),(c4=1)]->(class=1)  (p=884.0, n=0.0, m=2/4, heuristic_value=0.9981981688891336)
[(c2=5),(c4=5),(c1=5),(c3=5)]->(class=1)  (p=884.0, n=0.0, m=2/4, heuristic_value=0.9981981688891336)
[(c1=6),(c4=6),(c2=6),(c3=6)]->(class=1)  (p=884.0, n=0.0, m=2/4, heuristic_value=0.9981981688891336)
[(c2=2),(c3=2),(c1=2),(c4=2)]->(class=1)  (p=884.0, n=0.0, m=2/4, heuristic_value=0.9981981688891336)
[(c3=3),(c4=3),(c1=3),(c2=3)]->(class=1)  (p=884.0, n=0.0, m=2/4, heuristic_value=0.9981981688891336)
[(c1=1),(c1=4),(c1=6),(c2=1),(c2=2),(c2=5),(c3=2),(c3=3),(c3=4),(c4=3),(c4=5),(c4=6)]->(class=0)
                                          (p=480.0, n=0.0, m=4/12, heuristic_value=0.9970977726611563)
[(s1=2),(s2=2),(s3=2),(s4=2)]->(class=0)  (p=360.0, n=0.0, m=4/4, heuristic_value=0.9961383586648443)
[(s1=1),(s2=1),(s3=1),(s4=1)]->(class=0)  (p=360.0, n=0.0, m=4/4, heuristic_value=0.9961383586648443)
[(c1=1),(c1=4),(c1=6),(c2=1),(c2=5),(c3=3),(c3=4),(c4=3),(c4=5),(c4=6),(c4=2)]->(class=0)
                                          (p=288.0, n=0.0, m=4/11, heuristic_value=0.9951829010149089)
...
```

Figure 1: Best ten unique rules learned on the artificial pairs data set (using m=3 for the m-estimate). Every rule consists of a rule body in brackets with features concerning the rank (c_i for $i \in 1..4$) or suit (s_i for $i \in 1..4$) of the four cards, followed by an arrow and the predicted rule head (0 for no pair, 1 for pair). After each rule various heuristics are shown: The number of true and false positives p and n, the values used for the m-of-n concept in the format m/n, and finally the m-estimate of the rule.

4. Experiments

For the artificial data set, we use a special split between training and test set, for the real-world data sets, we use ten-fold-cross-validation instead. For all experiments, we use the m-estimate metric and tested eight different values between 0 and 100. The m-estimate value h_m of a rule r predicting class c has been proposed by Cestnik [10] and is calculated as

$$h_m(r) = \frac{r.p + m\frac{P}{P+N}}{r.p + r.n + m}, \tag{2}$$

where

m = a settable parameter in the range [0, +∞)
$r.p$ = the number of true positives of rule r
$r.n$ = the number of false positives of rule r
P = the number of examples with class = c
N = the number of examples with class $\neq c$.

Artificial data set. For the first part of the experiments, we reused a variant of the pairs data set presented in an earlier paper [11]. Each example consists of four cards, which in turn consist of a rank (ace, 2, 3, ..., queen, king) and a suit (clubs, spades, hearts, diamonds). Each card is therefor defined by a unique rank-suit-combination, e.g. "spades 7". In this paper, we use a smaller numeric subset of two suits $\{1, 2\}$ and six ranks $\{1..6\}$ to obtain 12 different cards, and generate all 11,880 combinations as examples. All examples where at least two of the ranks are equal, i.e., they contain a pair, are assigned to the positive class, all others to the negative class.

The negative examples are distributed evenly between training and test set. However, for the positive examples, we always pick one different pair combination for each rank that is hold back for the test set. For example, all pairs of 1s in the first two cards are retained for the test set, all other pairs of 1s are in the training set. As a consequence, state-of-the-art DNF rule learners like Lord are only capable to detect these five pair combinations and will miss the last possible combination since they are not able to generalize well enough.

Figure 1 shows the best ten rules learned by m-of-n Lord. Independent of the chosen value of m for the m-estimate, the first six rules are the optimal pair descriptions of the positive class. The algorithm detected the four relevant "rank"-features for each rank $\{1..6\}$ and correctly learned that if at least two these are true, the example will be positive. Thus, it could also generalize to the missing pair that was only available in the test set.

The m-of-n concepts learned for the negative class are interesting as well. Two of them are short and easy to understand: If four of four cards have the same suits, no pair can occur since we did not include duplicates. The remaining two are harder to grasp and result from the special split of training and test set: Since four of the eleven/twelve features have to be true, all ranks are preselected of those among the features, and the only pairs that can be built are those retained for the test set (e.g. $c_1 = 1$ and $c_2 = 1$).

Real-world data sets. The second part of the experiments was executed on real-world data sets. We used the same 29 UCI data sets as [12] and compared m-of-n Lord with the DNF rule learner regular Lord and the CNF rule learner k-CNF [12]. Surprisingly, m-of-n Lord could not compete at all with the other two rule learners. Compared with regular Lord, it lost on 24 data sets, tied on 3 and won on 2, with an average accuracy difference of 3 percentage points, independent of the choice of parameter m for the m-estimate.

Simple adjustments to the algorithm like ending with an additional specialization iteration or starting with a most-specific rule and first generalizing and then specializing it, did not increase the accuracy remarkably and generally even performed worse. Interestingly, ignoring the generalization step completely improved the performance on many data sets but also worsened its performance on the pairs data set drastically. This generalization step was also needed to achieve a promising performance on the similarly structured monks-2 benchmark data set: m-of-n LORD achieved 93% accuracy — 30% more than conventional rule learners.

5. Conclusion

In this paper we analyzed m-of-n concepts as a generalization of the logical conjunctive and disjunctive concepts learned by most state-of-the-art rule learners. We proposed a novel dynamic programming approach to learn m-of-n concepts as Boolean expressions, and use this technique to extend the rule learner LORD to learn m-of-n concepts as well. This learner reliably found the generalized model for the pairs data set, which can not be learned by state-of-the-art rule learners. However, it could not achieve a similar performance like these rule learners on general real-world data sets, so that a more sophisticated design of the algorithm is needed in the future.

6. Future Work

An even more general representation than m-of-n concepts are scoring systems, which assign a weight to every of the n features and have a flexible threshold that is usually greater than n. This form is even closer to perceptrons and can therefor be extracted even easier from neural networks.

However, scoring systems can also be generated in two possible ways in Boolean expressions. For example, to learn a scoring system with features $[a, b, c]$, weights [2, 1, 1] and a threshold of 3, m-of-n concepts can still be helpful. If generalization and specialization operations are allowed to also add features that are already contained in the concept again, we can use the generalization 2 of $[a, b, c] \rightarrow$ 3 of $[a, a, b, c]$ to obtain the mentioned scoring system. This can already be achieved by slightly adjusting Algorithm 1 to ignore `remaining_features` completely.

Another option to emulate "weightings" in Boolean expression are nested structures. M-of-n concepts could be learned in multiple layers, so that for the given example the deep concept 2 of $[a,$ 1 of $[b, c]]$ could be found.

References

[1] P. M. Murphy, M. J. Pazzani, Id2-of-3: Constructive induction of m-of-n concepts for discriminators in decision trees, in: Machine learning proceedings 1991, Elsevier, 1991, pp. 183–187.

[2] G. G. Towell, J. W. Shavlik, Extracting refined rules from knowledge-based neural networks, Machine learning 13 (1993) 71–101.

[3] R. Setiono, Extracting m-of-n rules from trained neural networks, IEEE Transactions on Neural Networks 11 (2000) 512–519.

[4] S. Odense, A. d'Avila Garcez, Extracting m of n rules from restricted boltzmann machines, in: Artificial Neural Networks and Machine Learning–ICANN 2017: 26th International Conference on Artificial Neural Networks, Alghero, Italy, September 11-14, 2017, Proceedings, Part II 26, Springer, 2017, pp. 120–127.

[5] F. Maire, A partial order for the m-of-n rule-extraction algorithm, IEEE Transactions on Neural Networks 8 (1997) 1542–1544.

[6] P. T. Baffes, R. Mooney, Symbolic revision of theories with m-of-n rules, 1993.

[7] P. Langley, Elements of machine learning, Morgan Kaufmann, 1996.

[8] V. Q. P. Huynh, J. Fürnkranz, F. Beck, Efficient learning of large sets of locally optimal classification rules, Machine Learning 112 (2023) 571–610.

[9] Z.-H. Deng, S.-L. Lv, Prepost+: An efficient n-lists-based algorithm for mining frequent itemsets via children–parent equivalence pruning, Expert Systems with Applications 42 (2015) 5424–5432.

[10] B. Cestnik, Estimating probabilities: A crucial task in Machine Learning, in: L. Aiello (Ed.), Proceedings of the 9th European Conference on Artificial Intelligence (ECAI-90), Pitman, Stockholm, Sweden, 1990, pp. 147–150.

[11] F. Beck, J. Fürnkranz, P. H. V. Quoc, On the incremental construction of deep rule theories, in: L. Ciencialová, M. Holena, R. Jajcay, T. Jajcayová, F. Mráz, D. Pardubská, M. Plátek (Eds.), Proceedings of the 22nd Conference Information Technologies - Applications and Theory (ITAT 2022), Zuberec, Slovakia, September 23-27, 2022, volume 3226 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 21–27. URL: https://ceur-ws.org/Vol-3226/paper2.pdf.

[12] A. Dries, L. De Raedt, S. Nijssen, Mining predictive k-CNF expressions, IEEE Transactions on Knowledge and Data Engineering 22 (2009) 743–748.